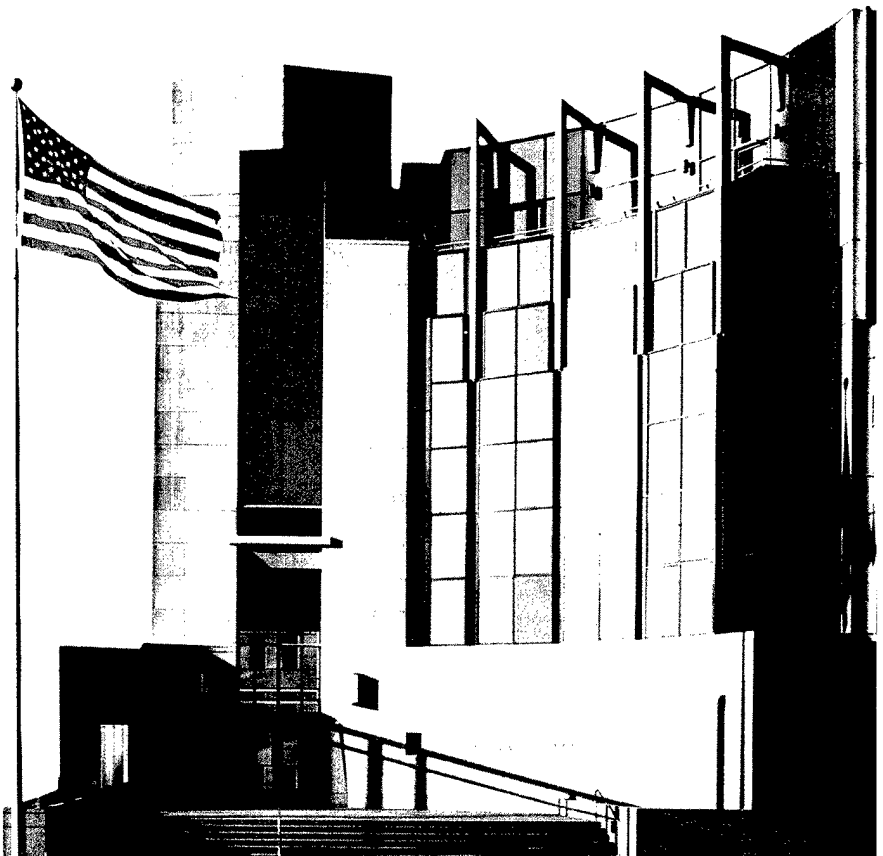
Carnegie Mellon
Software Engineering Institute

Illuminating the Fundamental Contributors to Software Architecture Quality

Felix Bachmann
Len Bass
Mark Klein

August 2002

TECHNICAL REPORT
CMU/SEI-2002-TR-025
ESC-TR-2002-025



Carnegie Mellon University does not discriminate and Carnegie Mellon University is required not to discriminate in admission, employment, or administration of its programs or activities on the basis of race, color, national origin, sex or handicap in violation of Title VI of the Civil Rights Act of 1964, Title IX of the Educational Amendments of 1972 and Section 504 of the Rehabilitation Act of 1973 or other federal, state, or local laws or executive orders.

In addition, Carnegie Mellon University does not discriminate in admission, employment or administration of its programs on the basis of religion, creed, ancestry, belief, age, veteran status, sexual orientation or in violation of federal, state, or local laws or executive orders. However, in the judgment of the Carnegie Mellon Human Relations Commission, the Department of Defense policy of "Don't ask, don't tell, don't pursue" excludes openly gay, lesbian and bisexual students from receiving ROTC scholarships or serving in the military. Nevertheless, all ROTC classes at Carnegie Mellon University are available to all students.

Inquiries concerning application of these statements should be directed to the Provost, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-6684 or the Vice President for Enrollment, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, telephone (412) 268-2056.

Obtain general information about Carnegie Mellon University by calling (412) 268-2000.

Carnegie Mellon
Software Engineering Institute

Pittsburgh, PA 15213-3890

Illuminating the Fundamental Contributors to Software Architecture Quality

CMU/SEI-2002-TR-025
ESC-TR-2002-025

Felix Bachmann
Len Bass
Mark Klein

August 2002

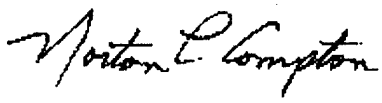
Architecture Tradeoff Analysis Initiative

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Acknowledgements

We would like to thank Judy Stafford and Gary Chastek for their thoughtful review of an earlier draft of this report, Bill Wood for providing the marshaling example, Anna Liu for general feedback, Robert Nord for participating in discussions that led to this report, and Linda Northrop for advice about its content.

Abstract

An architectural tactic is a design decision that helps achieve a specific quality-attribute response. Such a tactic must be motivated by a quality-attribute analysis model. This report presents the basic concepts of analysis models for two quality attributes—modifiability and performance, identifies a collection of tactics that can be used to control responses within those models, and discusses how to analyze the models in terms of these tactics.

This report also describes how to interpret architectural designs in terms of analysis models and how to apply those models to specific architectures. In addition, it presents the analysis of several different architectural patterns taken from current literature.

1 Introduction

A software architecture is widely recognized as a keystone artifact of software engineering. A system's software architecture is the key to communicating and comprehending that system. It is therefore natural to ask, what are the key principles that exert a dominant influence on the "shape of an architecture?" And with the answer to that question in hand, it is also natural to ask, how can those principles be exploited by both a design methodology to produce suitable architectural designs, and an analysis methodology to analyze the suitability of those designs?

The Product Line Systems Program at the Software Engineering Institute (SEI) has been working on these questions for several years. One of the key principles that direct our work is that the quality-attribute requirements (such as performance, security, modifiability, reliability, and usability) exert a dominant influence on the "shape" of a software architecture. Based on this principle, we have developed methods for analyzing and designing software architectures. These methods include the Architecture Tradeoff Analysis Method[SM] (ATAM[SM]), the Quality Attribute Workshop (QAW), and the Attribute-Driven Design (ADD) method.

We verify this principle every time we conduct an architecture evaluation using the ATAM or an architectural design using the ADD method. With minimal domain knowledge, a team armed with relevant attribute knowledge can find architectural risks using the ATAM or make architectural design decisions to realize quality-attribute goals. However, while these methods efficiently orchestrate the use of quality-attribute knowledge, they don't embody that knowledge. To properly execute those methods, analysts and designers must be armed with quality-attribute knowledge and experience.

The goal of this report is to describe how quality attributes exert influence over architectural design, how these influences can be codified, and how these notions can be used to analyze architectures. The central notion that we introduce in this report is the *architectural tactic.* Just as design patterns and architectural patterns are carriers of design knowledge, we envision a collection of architectural tactics as the carriers of quality-attribute-based design knowledge.

[SM] Architecture Tradeoff Analysis Method and ATAM are service marks of Carnegie Mellon University.

The difference between architectural tactics and architectural patterns is that quality-attribute analytic models directly motivate architectural tactics. For example, for years, queuing and scheduling theories have been used for creating performance models, and Markov chains have been used to create reliability models. These models provide a formal vehicle for reasoning about the relationship between the various properties of an architectural design.

For example, it is common to specify average latency requirements for some aspects of a system. However, average latency is not directly controllable. To achieve a specified average latency, design properties—such as the size and number of Web servers, the number of threads per server, execution times, and thread priorities—must be decided. Performance models describe the relationship between these settable properties and dependent properties that are controllable only indirectly, such as average latency.

We are now ready to define *architectural tactic*:

> An architectural tactic is a design decision that helps achieve a specific quality-attribute response and that is motivated by a quality-attribute analysis model.

In the above example, the size and number of servers, and the management of concurrency on a server are all design decisions that can be made by the architect. Average latency is a response of the system to a stimulus, such as a user request. Setting the number of servers and managing concurrency on the server will have a dramatic impact on the average latency. This relationship is evident from queuing and scheduling models, which are analytic models for performance.

Our definition of tactic has the following consequences:

- An architectural tactic is concerned with the relationship between design decisions and a quality-attribute response. This response is usually something that would be specified as a requirement (e.g., an average latency requirement). Therefore architectural tactics (by definition) are points of leverage for achieving quality-attribute requirements. Consequently, codifying architectural tactics will involve articulating rules for making design decisions that control how and why the response varies.
- Analytic models for the various quality attributes allow us to identify design decisions that offer leverage for achieving quality-attribute requirements. The analytic models also offer a reasoning framework for explaining how changes in the design decisions affect the response. Architectural tactics "live" within this reasoning framework.
- While architectural tactics are motivated from an analytic perspective, they have specific realizations in an architectural design. For example, while a queuing model might only require the average execution time as input to the model, there are many sources of execution time that need to be derived from an architectural design. Or, while a queuing model might be concerned only with how average latency varies as a function of the number of servers, in the architectural model, servers might be Web servers in one case and processors of a multiprocessor in another. There is a many-to-one relationship be-

tween the analytic model and the corresponding architectural realizations. Consequently, codifying architectural tactics involves articulating some of the possible mappings from an architectural model to an analytic quality-attribute model.

While the examples we use above draw on formal models such as queuing models, we also consider quality attributes with less formal underpinnings. All that we require is some analytic reasoning framework that describes the relationship between controllable aspects of a design and the quality-attribute response. In this report, we consider one attribute with formal underpinnings—performance—and another attribute with less formal underpinnings—modifiability.

Architectural tactics exist at various levels of abstraction just as design decisions do. In the above example, we call the task of determining the scheduling policy, which is a design decision, a tactic. Scheduling by a first-in, first-out (FIFO) policy is also a design decision: it is a refinement of the original tactic. Thus, both of these design decisions represent tactics.

In general, we think that these ideas hold the potential for having a dramatic influence on architectural design and analysis. And they might be of interest to the aspect-oriented design community, since the tactics they involve allow (at least in concept, if not yet in practice) the weaving in of design decisions that will provide particular quality-attribute responses.

In this report, we restrict ourselves to using tactics for the analysis of a few aspects of the modifiability and performance quality attributes. Our intention is to introduce these ideas as soon as possible and receive community feedback. In the future, we intend to discuss the application of tactics to architectural design; add to our coverage of modifiability and performance; and analyze tactics for additional quality attributes.

We begin this report by offering background that compares our current approach to our previous attempts to codify the relationship between quality-attribute responses and design decisions. The remainder of this report discusses the following topics as they relate to the modifiability and performance quality attributes: general scenarios that specify quality-attribute-specific stimuli and responses, and the essentials of analyzing the attributes (Sections 3 and 4), the architectural tactics for each attribute, how to analyze each tactic for the desired general scenarios, and applying the analysis to two example patterns (Section 5). Finally, we offer our conclusions (Section 6).

We assume that the reader is familiar with the concept of general scenarios and with the general scenarios used to characterize modifiability and performance [Bass 01].

2 Background and Tactics Overview

2.1 Background

Our first attempt at codifying attribute knowledge was the generation of Attribute-Based Architectural Styles (ABASs) [Kazman 99]. An ABAS is a description of an architectural style packaged together with an analysis of that style for one quality attribute.

Three elements of this formulation survive. First, packaging analysis and architectural elements (of some sort) is still a key goal. Secondly, performing the analysis for one attribute per tactic is still our approach. Tradeoffs among attributes are identified during the analysis process and either reported (during an evaluation) or adjusted by the designer (during a design effort). Thirdly, the distinction between the analysis model and the architectural model that we discussed earlier requires a mapping between them.

The main problem with the ABAS notion is the vast number of possible ABASs and architectural patterns. Each architectural pattern (or style) requires an ABAS for each attribute. On the other hand, we expect there to be relatively few architectural tactics.

Our next attempt at the codification was to identify attribute primitives [Bachmann 00]. The premise behind this work was that for each quality attribute, there existed a relatively small collection of primitive architectural elements. Attribute primitives were to be the building blocks for ABASs, thereby solving the problem of having a huge number of ABASs. Identifying and analyzing these primitives and understanding how to compose them would enable the evaluator or designer to package the analysis with the architectural element.

One element of this formulation survives. We still believe that there are architectural building blocks (that are smaller than styles), the analysis of which contributes to the analysis of a style and subsequently to a system. We also believe that the number of such building blocks is relatively small.

Our problem with attribute primitives is with the requirement that the architectural building block be primitive. Attempting to identify primitives leads to questions of whether what has been identified is indeed a primitive. For example, when considering a publish-subscribe style, are the primitives the publishers, subscribers, distribution mechanism, and registration mechanisms? the publishers, subscribers, and distribution mechanism? or just the distribu-

tion mechanism? Any of these choices can be defended or attacked. Useful building blocks needn't be primitive; the search for primitives was a digression from our main goal.

2.2 Tactics Overview

The current formulation for combining analysis with architectural building blocks is:

- For each quality attribute, enumerate analytic building blocks (called tactics) that are used to control it.
- Associated with each tactic is an analysis approach for its attribute.
- The analysis results drive the design decisions.
- Architectural patterns or systems are applications of one or more of these tactics.
- The analysis of tactics leads to a method for analyzing the architectural patterns or systems.

Consider the following simplified architectural patterns:

- a three-tiered architecture. This was defined to support modifiability and in Section 3, we show why. We also show how the tactic for hiding information limits the effect of a change to a module and how the tactic for breaking the dependency chain limits the ripple effect of a change that can't be confined within a module.
- a multiple-processor server associated with an e-commerce site. Multiprocessor configurations are established to make applications perform better (e.g., with shorter average response times). In Section 4, we show how the tactics for increasing the physical concurrency and for balancing resource allocation, with their associated analyses, can influence the application's response time. These analyses are used to determine the number of processors and the balancing algorithms for a certain arrival pattern of requests.

Both of these examples exemplify our goal to make an initial analysis with simplifying assumptions readily available. While this allows coarse-grained analysis, it also makes performing the analysis possible by people without sophisticated knowledge of analysis models.

Our interest in this report is in *analysis*, that is, the testing of a design against a set of criteria. The design process consists of the designer generating a hypothesis and then testing that hypothesis against some set of criteria. If the hypothesis does not pass the test, another hypothesis is generated and tested. Clearly, the generation of a design hypothesis is not a random process, and highly skilled designers generate better hypotheses than less skilled designers do. The tactics we discuss have a role in the generation of hypotheses, but that role is not discussed in this report. We focus only on the testing of a design against criteria.

The key behind identifying tactics is that they originate in analysis, in this case, in performance and modifiability analysis. The essence of analysis is that it offers very compelling (and sometimes formal) arguments for how key aspects of an architecture affect attribute goals. Tactics address those important aspects.

The power of tactics is that there is a many-to-one relationship between the various architectural situations and analysis. In other words, many architectural situations can be reduced to the same analytic model. This is why there is leverage in codifying tactics. They allow us to focus on what is important and offer guidance as to how to find and control these important aspects of architecture.

We now turn to the first attribute that we discuss in-depth—modifiability.

3 Modifiability

Modifiability is about the cost of making changes. Attempting to make an architecture modifiable means to localize changes in a hopefully small area of it. One of an architect's worst nightmares is that a small change in one part of the architecture unintentionally affects many other parts.

Changes have three aspects:

1. *what* is being changed—Is it the user interface or the operating system (OS)? Is it an additional feature for a user or is a new peripheral device being connected? Does the system have to be scaled up or does it need to be able to offer higher quality services?
2. *who* should make the change—Should the change be made by the developer via changing the code? by the system administrator via changing a configuration file? or by users via customizable system features?
3. *when* the change is made—Is the change made during development? when the system is initially configured? during installation? at start-up? or while the system is running?

In this report, we assume that a change or set of changes arrives at the developer's desk during development. For this reason, we confine our attention to developers and do not cover changes made after deployment by a system administrator or an end user. We do not cover deployment time or the tactics associated with deferring binding time either.

3.1 Specifying Modifiability Requirements

We use scenarios to specify attribute requirements. Modifiability requirements are specified with change scenarios. A scenario consists of at least a stimulus and a response. In general, the stimuli for change scenarios are anticipated changes to the system, and the response is the expected cost of making those changes (measured in terms of the number of modules that must be modified). Changes can be classified according to

- probability—How likely is it that a certain change will occur? For example, how likely is it that the OS has to be exchanged for another?
- frequency—How often will a certain change occur? For example, how often will new user features or peripheral devices need to be added?

- dependence—Is the change connected to another different type of change? For example, does adding a new peripheral device require the addition or modification of user features?

The system's response to changes is usually measured in terms of costs and/or time. The following questions need to be answered: How difficult will it be to make a certain change? How long will it take to make it? How much will it cost?

3.2 Primary Elements Affecting Modifiability

Two key factors of the model are used for analyzing modifiability:

1. the module itself. A module is an implementation unit of software that provides a coherent unit of functionality. Each module has a collection of interfaces and a set of responsibilities that define the tasks it must perform. Other modules rely on the module's handling of those tasks.
2. what the module depends on. For example, module B depends on module A if a change to module A, in turn, requires a change to module B.

We are fundamentally interested in how long it will take or how much it will cost to make a set of changes. Since that is very difficult to predict, we, instead, try to measure the impact of the changes on the existing design. To do so, we first assign a weight to each module that reflects its relative difficulty to modify. For example, a module that is easy to change would be assigned the number 1, whereas a module that is difficult to change would be assigned the number 5. Then we add up the weights of each module that must be changed in order to implement a set of changes. The total gives us the set's impact—the higher the number, the greater the impact.

3.3 Example Showing Contributors to Modifiability

In this section, we briefly describe a typical scenario and then scrutinize it to determine common factors that contribute to modifiability. These factors are important to achieving modifiability-response requirements and therefore must be controlled. In Section 3.4, we offer a list of modifiability tactics that can be used to control these factors.

In the example shown in Figure 1, we assume that a system is divided into three layers. The top layer, the user interface, has the responsibility of accepting commands from the user and then displaying their results. The middle layer, the application, provides services for the user, and the bottom layer, the hardware abstraction, has the responsibility of communicating with connected devices such as sensors and actuators.

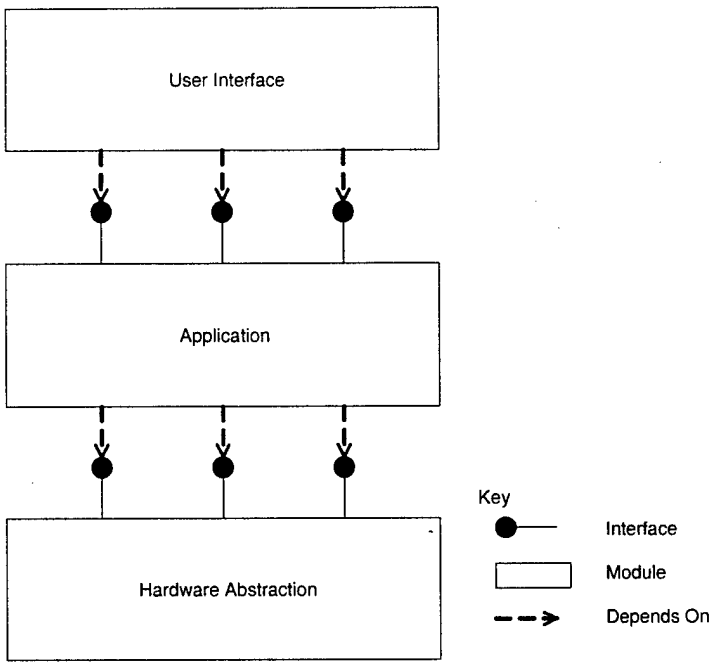


Figure 1: A Layered Architecture Example

Each layer has been implemented as a separate set of modules with an application program interface (API) that is used only by the adjacent higher layer. In terms of our modifiability model, this means that each layer has, at the very least, several types of dependencies on the layer below. In Section 3.5.2, we enumerate various types of dependencies and show that their relationships are actually more complicated.

The general flow of events in this scenario is that a user issues a command to the application. Then, the application executes the command with the help of the hardware abstraction and displays the results to the user.

3.3.1 Factors Contributing to Modifiability

Separating functionality into different layers supports certain types of changes. Changing the hardware would affect the hardware abstraction but hopefully would have only minimal effects on the application and the user interface. Changing the application's functionality should not affect the user interface or the hardware abstraction, and changing the user interface should not affect the application or the hardware abstraction.

Factors that contribute to modifiability include

- Responsibilities are assigned to the layers to support the division into the user interface, application, and hardware abstraction. As long as they can fit into these categories, requested changes can be made more easily.

  Access to a service through an interface hides details of the module from other modules. As long as a change affects only those details (and doesn't affect the actual interface), all modules using the changed module remain unaffected. However, if a change affects a module's interface, all modules using that interface must be changed as well.

- How the interface is defined, what is made visible, and what is hidden all have a strong influence on modifiability.

  If a lower layer has to be changed (e.g., the hardware abstraction) and the change will become visible in the interface, the next higher layer (e.g., the application) also has to be adapted. Whether this secondary change will be visible in the user interface depends on how the secondary change was made.

- The middle layer can act as an intermediary to prevent changes to the lower layer from being propagated to the top layer.

We now present the tactics used to control the number of modules impacted by a set of changes. These tactics are motivated by three factors: locality, visibility, and the interdependency of changes.

3.4 Modifiability Tactics

The tactics for affecting modifiability are organized into three sets: (1) those that localize expected modifications, (2) those that restrict the visibility of responsibilities, and (3) those that prevent the ripple effect. Each set is described below.

3.4.1 Tactics for Localizing Expected Modifications

The responsibilities that are assigned to modules greatly influence the cost of making a change. Depending on how the assignment was done, a specific change can affect either a single module or multiple ones. The goal of this set of tactics is to affect as few modules as possible with a single change by presenting guidelines for how responsibilities are assigned. Possible tactics include

- **Maintain semantic coherence.** Semantic coherence refers to the relationships between a module's responsibilities. The goal here, that of ensuring that all of a module's responsibilities work together without excessive reliance on other modules, is achieved by choosing responsibilities that have some sort of semantic coherence. Doing so binds re-

sponsibilities that are likely to be affected by a change. As discussed in Section 3.5.1, knowing whether semantic coherence is achieved comes from examining the likely changes and determining whether changes of the same type impact the same set of modules. One subtactic of this tactic is isolating common services. Common services are those used by multiple modules. They can be isolated into libraries invoked by other modules, classes that are inherited, or code that provides a template for instantiation services. Providing a common service is a form of semantic coherence, since it requires the services to coordinate their activities.

- **Isolate the expected change.** Separating the responsibilities that are *likely* to change from those that are *unlikely* to change separates an architecture into fixed and variant parts. This enables more design effort to be devoted to making modules as easy to change as possible.
- **Raise the abstraction level.** Raising the abstraction level, and thus making a module more general, allows that module to calculate a broader range of functions based on input. This can be as simple as using constants as input parameters or as complicated as implementing the module as an interpreter and using the input parameters as a program in the interpreter's language. The more general a module is, the more likely it will be that requested changes can be made by adjusting the input (rather than modifying the module).

3.4.2 Tactics for Restricting the Visibility of Responsibilities

If a module is affected by a change, it is important to know whether that change will become visible outside of the module. If it will, changes to other modules will most likely be required. Possible tactics for restricting visibility include

- **Hide information.** This tactic is based on dividing a module's responsibilities into two categories: public and private. Public responsibilities are those that are visible from both inside and outside the module. Private responsibilities are those that are visible only from inside the modules. The goal of this tactic is to limit the public responsibilities and make them visible through an interface. As discussed in Section 3.5, the proof of whether hidden information is effective is in the visibility of changes outside the affected module. Changes to the affected module that are visible from other modules have not been hidden.
- **Maintain existing interfaces.** This tactic is based on keeping interfaces constant across a particular change. That is, the module developer will maintain the old identity, syntax, and semantics of an existing interface even if the modification changes them. Typically, the developer does this by either adding a new interface or assigning a version number to an existing interface for the changed module.
- **Separate the interface from the implementation.** This tactic allows the realization of the implementation later in the development process than the interface specification. It also allows the interface to be specified at a higher level of abstraction than the implementation. Rather than embodying the abstraction into the implementation (as in the tactic described in Section 3.4.1 for raising the abstraction level), this tactic will have an interface with a higher level of abstraction than the actual implementation. The idea is that the implementation can be changed fairly radically without affecting the calling program.

3.4.3 Tactics for Preventing the Ripple Affect

A ripple effect from a modification is the necessity for making changes to modules that are not directly affected by that modification. This necessity occurs because of a dependency between the module that is directly affected and another module that is dependent on it. Possible tactics for preventing the ripple affect include

- **Break the dependency chain.** This tactic refers to the use of an intermediary to keep one module from being dependent on another, and therefore to break the dependency chain. Typically, the name of the intermediate depends on the type of dependency it breaks. The following example of intermediates can also be seen as tactics:
  - **Use a name server.** A name server breaks a dependency on the runtime location of a module.
  - **Use a virtual machine.** Virtual machines break dependencies on computations specific to a particular situation. Examples of virtual machines are the hardware abstraction layer in our example, the Factory pattern [Gamma 95], and the controller in the Presentation, Abstraction, and Controller pattern [Buschmann 96].
  - **Use a publish-subscribe pattern.** A publish-subscribe pattern breaks a dependency of a data consumer on the identity of the data producer.
  - **Use a repository.** A repository can be used to break two different types of dependencies: (1) the dependence of a data consumer on the identity of the publisher (as in the Publish-Subscribe pattern), or (2) the dependency of the data consumer on the data's syntax. Modern repositories allow the consumer to specify the type in which data is presented to them, regardless of the data's type in the repository.
  - **Use a dynamic scheduling algorithm.** Some scheduling algorithms, such as semantic-importance-based scheduling, guarantee that deadlines will be achieved within certain restrictions. To a certain degree, this can be used to compensate for a possibly higher utilization of resources due to a change, therefore alleviating a dependence on resource usage.
- **Make the data self-identifying.** Tagging the data with identification information, such as sequence number (as in network protocols), syntax descriptions (as in some languages with dynamic runtime typing), or identity (as in free-form parameter invocation), will break dependencies on either sequencing or syntax.

3.5 Modifiability Analysis

We divide our analysis into two portions: (1) determining how well the responsibilities were allocated to localize the expected modifications, and (2) determining how well information was hidden and whether there are any ripples.

3.5.1 Analysis for the Allocation of Responsibilities

We assume for our analysis that the needed changes fall into two categories:

1. those that are variants of each other
2. those that affect different functionality

Ideally, a specific type of change requires the adaptation of only the module that contains the responsibilities that must be changed. This would greatly limit the side effects of a change, especially if the tactic for hiding information is used for the module. Therefore, this analysis determines how well distributing responsibilities to different modules supports different changes. If two changes are variants, they should impact exactly the same modules; if they affect different functionality, they should impact different modules.

The analysis proceeds by determining how the modifications determined by change scenarios are distributed across the modules. The goals of this analysis are that similar change scenarios require modification of the same modules and that different change scenarios do not overlap in the modules to which they require changes. This depends completely on how responsibilities have been assigned to the modules.

To analyze how responsibilities are allocated, first determine which module's responsibilities are modified by each change scenario. Doing so provides an estimate of the cost of that change, as previously discussed. Next, determine the number of modules that are affected by more than one type of modification. The cost of making modifications to a module that is impacted by different types of modifications is larger than for a module that is not. This is due to the side effects caused when the responsibilities of many types of modifications interact.

In light of this analysis, examining the tactics for localizing expected changes explains why those tactics can control the expected response. If the responsibilities are allocated according to semantic coherence, the analysis will determine whether the correct criteria for semantic coherence were chosen, and whether the responsibilities were allocated according to that choice. If the responsibilities are allocated according to the expected changes, the analysis will determine whether the allocation was performed correctly. If the level of abstraction is sufficiently high, the projected modification can be accomplished through a change of input to the module rather than through a change of responsibilities.

3.5.2 Dependencies

The remainder of the analysis requires understanding the conditions under which multiple modules might be affected by a modification. This, in turn, requires understanding dependencies among modules. If a modification is made to some aspect of module A and module B is dependent, in some sense, on module A, then module B may have to be modified to accommodate the modification to module A.

3.5.2.1 Types of Dependencies

There are eight types of dependencies:

1. syntax dependencies
   - of data. In order for module B to compile (or execute) correctly, the type (or format) of the data that is produced by module A and consumed by module B must be consistent with the type (or format) of data assumed by module B.
   - of service. In order for module B to compile and execute correctly, the signature of services provided by module A and invoked by module B must be consistent with the assumptions of module B.
2. semantics dependencies
   - of data. In order for module B to execute correctly, the semantics of the data that is produced by module A and consumed by module B must be consistent with the assumptions of module B.
   - of service. In order for module B to execute correctly, the semantics of the services that are produced by module A and consumed by module B must be consistent with the assumptions of module B.
3. sequence-of-use dependencies
   - for data. In order for module B to execute correctly, module A must produce data in the sequence assumed by module B (e.g., header before body).
   - for control. In order for module B to execute correctly, module A must have executed previously within certain timing constraints (e.g., module A must execute no earlier than five milliseconds [ms] before module B executes).
4. interface identity dependencies
   Module A may have multiple interfaces. In order for module B to compile and execute correctly, the identity (name or handle) of the interface must be consistent with the assumptions of module B.
5. runtime location dependencies
   In order for module B to execute correctly, the runtime location of module A must be consistent with the assumptions of module B. For example, module B may assume that module A is located in a different process on the same processor, so it uses the services of module A and sends messages to its process.
6. quality-of-service or -data dependencies
   These dependencies involve the service or data provided by a module. In order for module B to execute correctly, some property involving the quality of the data or service provided by module A must be consistent with module B's assumptions. For example, data provided by a particular sensor read by module A must have certain accuracy in order for the algorithms of module B to work correctly.
7. existence-of-module dependencies
   In order for module B to execute correctly, module A must exist. For example, module B is requesting a service from an object that is created dynamically and therefore may or may not exist.
8. resource behavior dependencies
   In order for module B to execute correctly, the resource behavior of module A must be consistent with module B's assumptions. This behavior can involve either the resource

usage of module A (e.g., module A uses the same memory as module B) or resource ownership (e.g., module B reserves a resource and assumes that module A does not compete for it).

3.5.2.2 Special Characteristics of Dependencies

Special characteristics of dependencies include

- Dependencies are symmetric.
  Notice how dependencies are defined. If module B makes an assumption about module A and module A changes, module B must also be changed in order for the system to continue operating properly. The same is true if the assumptions are changed. That is, if an assumption that module B makes about module A changes, module A must also be changed. Changes to one module affect the other; therefore, the various types of dependencies always exist in both directions.
- Data- and service-semantic dependencies cannot be broken with intermediaries.
  For example, suppose that a module depends on receiving the address of a person. No intermediary can create an address in data where none existed before. Although the dependency cannot be broken, it could be weakened by using abstractions. For example, if the home address is not required, the person's office address would suffice.
- Dependencies are independent of specific changes.
  The dependencies we identify in the analysis are independent of specific modifications to the modules. This leads to a conservative dependency analysis, that is, the determination of when module A could be forced to change as a result of changes to module B. The specifics need to be determined in light of specific change scenarios.

3.5.2.3 Notation for Representing Dependencies

In this report, we represent dependencies through tables such as Table 1.

	Type of Change										
	Data Syntax	Data Semantics	Service Syntax	Service Semantics	Sequencing (Data)	Sequencing (Control)	Identity of an Interface	Existence	Location at Runtime	Quality of Service or Data	Resource Behavior
Propagation between modules A and B	+	+	+	+	-	-	+	+	+	-	-

Table 1: How Dependencies Are Represented for Modules A and B

This table shows the types of dependencies between two modules—A and B. A plus sign (+) indicates that the change (e.g., data syntax or resource behavior) may propagate between the modules. A minus sign (-) indicates that the dependency does not exist, so no propagation will occur. A plus sign does not guarantee that propagation will occur, only that it may. Whether the change actually propagates depends on the specific change that is made.

A filled-in dependency table represents the dependency analysis for a particular system. Three techniques for filling in these tables include

- using the brute-force method
- using transitive closure to determine when indirect propagation is possible
- using optimization techniques to reduce the amount of work required to fill in the dependency tables

Each technique is described below.

Using the Brute-Force Method

This involves these steps:

1. Enumerate all pairs of modules in the table.
2. For each pair of modules, consider whether a change of the type indicated in the column heading to one of the modules would require a change in the other module. The reason why a change would not propagate must be based on the implementation of one or more tactics, for example, if the module being changed contains hidden implementation information that keeps the modification from propagating. Another example would be if the data or control path from module A to module B goes through module C, and module C has implemented a tactic to break a dependency chain for the type of modification being considered.

Using Transitive Closure

Computing the transitive closure of the dependency matrix helps determine whether indirect propagation is possible between modules not linked directly. Figure 3 shows three modules A, B, and C and the data flow among them. In this sample data flow, data-syntax changes do not propagate *directly* from module A to module B, but they might propagate *indirectly* to module B through module C. This propagation could be prevented if module C uses the appropriate tactic, even if there are data-syntax dependencies between modules A and C and between modules B and C.

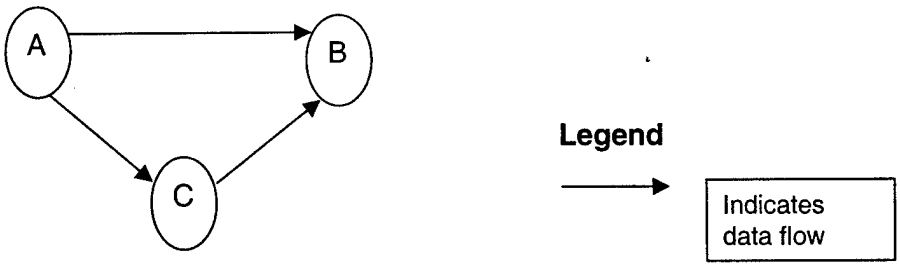


Figure 2: Data Flow Diagram Showing that Changes to Module A Might Propagate Indirectly to Module B

Using transitive closure involves these steps:

1. Fill in the dependency table using the brute-force method.
2. Determine where a change might propagate indirectly to another module.
3. In the cell for that type of change, indicate where propagation might occur by replacing the minus sign with a plus sign.

Using Optimization Techniques

Using optimization techniques can reduce the amount of work involved in filling in dependency tables by reducing the number of modules that actually need to be considered. These techniques consider the decomposition of one module into other modules, where dependencies are inherited. For example, if module A decomposes into modules B and C, modules B and C automatically inherit module A's dependencies. Likewise, if module A' decomposes into modules B' and C', a dependency will exist between modules B and B' *only* if such a dependency exists between modules A and A'.

Using optimization techniques involves these steps:

1. Fill in the dependency table using the brute-force method on the aggregated modules.
2. Determine whether propagation is possible based on whether the modules are decompositions of other modules. For any that are, look at the originating modules. If those modules will propagate the changes, so will the decomposed modules.
3. For any decomposed modules, make sure they have the same notations (plus or minus signs) as their originating modules do.

3.5.3 Observations About the Dependency Analysis

Observe how the dependency table is used during a modifiability analysis. After the table has been filled in, the architect considers a change scenario that will impact particular modules and their properties, as we discussed in Section 3.5.1. The architect looks at each module or property change resulting from the scenario and then indicates, using plus signs, which other modules it might affect. The architect must then reason about the extent of the change to the affected modules and consider it when determining the cost of the change.

Also observe how the tactics are related to the analysis. Each plus sign in the dependency table indicates that there were no tactics in place to prevent propagation of the change. So using modifiability tactics during the design process reduces the number of plus signs in the table, indicating a reduced chance of change propagation. If architects believe that the cost of a modification is too high, they can implement a tactic (or multiple tactics) so that a modification's impact is restricted. Then they can modify the dependency table accordingly. For example, if there is a dependency on the runtime location, the architect might decide to use a name server to break the dependency chain.

4 Performance

Performance is the ability of a system to allocate resources to service requests in a manner that will

- satisfy timing requirements and
- provide various levels of service (that depend directly on the amount of resources granted)

in the presence of

- competing requests,
- varying demand, and
- varying resource availability

Waiting arbitrarily or unpredictably long for the results of a computation is never suitable. While timing-related behavior is not always the dominant architectural driver, architecting with performance in mind is always appropriate.

4.1 Specifying Performance Requirements

Performance requirements are specified with performance scenarios that indicate the stimuli to the system and the required responses of the system to those stimuli. Performance stimuli are event streams (internal or external) that the system has to process. Event streams can be classified as follows:

- periodic—Interarrival intervals are equal.
- stochastic—Events arrive according to some probabilistic distribution.
- sporadic—Interarrival intervals can be no smaller than a specified minimum.

The response of a performance scenario is usually specified in terms of latency and throughput requirements: those for minimizing jitter; ensuring that precedence relations are adhered to; and providing spare capacity for growth.

Systems also need to be cognizant of the semantic importance of various functions so that in overload situations, when the demand cannot be satisfied, less important work is dropped to reduce resource demand, while more important work is predictably sustained.

While there are many dimensions to performance, latency is always important, and understanding the factors that influence latency sheds light on other facets of performance such as throughput. Therefore, we focus on latency in this report.

4.2 Primary Elements Affecting Performance

Latency is the time it takes a system to respond to a stimulus including the time spent waiting to get access to the resource and actually using it. The primary elements that affect latency, shown in a simple notation in Figure 4, include

- resources—physical units, usually having limited capacity, that provide services. Resource properties to consider include
  - Does the resource allow its usage to be preempted?
  - How powerful is the resource? For example, what is its central processing unit (CPU) clock speed or network bandwidth?
  - If multiple resources are required, are they needed one after the other or simultaneously?
- demand for resources—generated by stimuli that we refer to as *events*. An event is a request to perform a computation—or more generally, a request to use a specific resource). Examples of events include the arrival of a message, the expiration of a time interval (as signaled by a timer interrupt), and the detection of a significant change of state in the system's environment. We refer to a sequence of events that occurs over time as a *stream of events* (or simply as a stream). It is appropriate to think of a collection of events as a stream when their aggregate behavior can be described in terms of characteristics of their interarrival and execution times, such as a request from a single user or multiple users. Some properties of demand to consider are
  - How many streams of events are there?
  - What is the nature of the interarrival intervals between events in a stream?
  - How long does it take to process an event on each resource?
- arbitration—managing competing demands for resources according to certain rules. When multiple events require attention at the same time, the system must decide which one to address first and, in general, how to share the resource. One form of arbitration is the serialization of competing requests for resources that have limitations, such as being able to process only one request at a time. Some properties of arbitration to consider are
  - Which of several waiting requests should the resource grant?
  - How are the requests distributed between multiple resources?
  - How does the system react to overload conditions?
  - How many requests can wait for a resource to become available?
  - How long do requests have to wait on average for a resource?

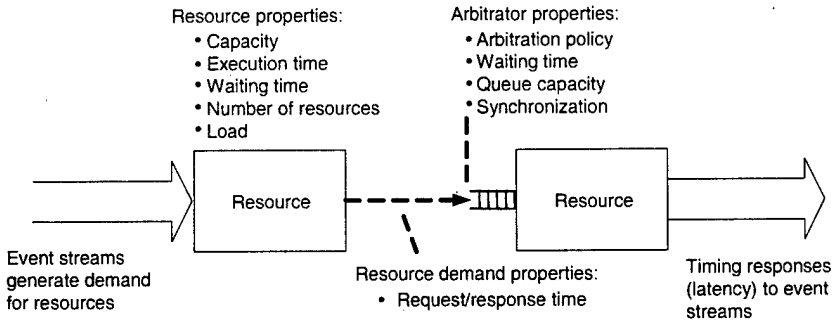


Figure 3: A Simple Notation for a Performance Model

4.3 Example Illuminating Contributors to Latency

In this section, we describe a typical scenario briefly and then scrutinize it to determine the common factors that contribute to latency. These factors are inimical to achieving latency requirements and therefore must be controlled. In the next section, we offer a list of performance tactics that can be used to control these factors.

4.3.1 Client/Server Scenario

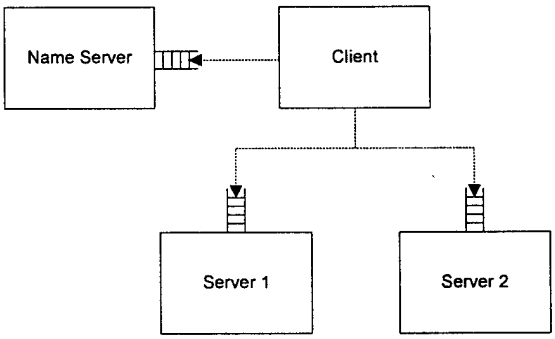


Figure 4: Analytic Model to Determine Latency for the Client/Server Example

In the example shown in Figure 4, a client invokes a service request to two independent servers. The locations of those servers are not known to the client, so they must be determined through a name server. The client sends a message to the first server via the middleware on both the client's processor and the servers' processors. After being "marshaled" by the middleware in the client's processor into a form that is recognizable by the middleware on the servers' processors, that message is copied into a message buffer with limited storage. If there is insufficient storage space available, the client will be blocked, and the message won't be sent over the communications network until it can be copied into the buffer. When a message arrives at the server, it is once again placed into a message buffer with limited storage. Next, the message is queued to be processed by the server and eventually serviced by the server. Then, the reply is sent back to the client by reversing the processes described

above. After some additional computation, the client sent its service request to the second server, which repeats the chain of execution described above.

4.3.2 Factors Contributing to Latency

Generally speaking, when an event occurs, it has to wait until the resource is ready to process the request. A resource might not be ready because it has failed, is initializing, or is not in the proper state to respond. However, often a resource is not ready to respond to one event simply because it is busy responding to other events. The arbitration strategy has a significant influence on waiting time. Therefore, when thinking about latency, it is important to consider both execution time (and its associated overhead) and the sources of waiting time that include

- an event that is queued behind other events in a queue
- a process that needs to access shared data, requires mutually exclusive access, and is being accessed by another process
- a resource that requires other resources to perform some work, such as responding to an input/output (I/O) request
- a process that is delivering the requested service and that has to share the processor with other processes. In this case, the execution might be interrupted while a higher priority process executes.

In some situations, it is useful to think of waiting time in terms of preemption time and blocking time. Preemption time is the time a response must wait while higher priority responses are executing. Blocking time is the time a higher priority response must wait while a lower priority process has use of a non-preemptable resource.

Consequently, the total latency for a resource to respond to an event comprises execution time, overhead, preemption time, blocking time, transfer time, and time due to serializing the computation. Note that our goal here is not to offer a taxonomy of different types of time, but rather to offer a starting point for considering the various contributors to latency. We use the scenario illustrated in Figure 4 when we discuss these contributors below. They provide motivation for the tactics enumerated in Section 4.4.

1. The serialization of requests leads to idle resources when physical concurrency can be exploited.

   If the client requests in our example are made synchronously, the total latency will be at least the sum of the latencies associated with each server request. If the requests are made asynchronously, it is possible that portions of the request can execute physically and concurrently on their respective servers leading to latency that is less than that sum.

   On the other hand, if there are precedence constraints in the response, physical concurrency may not help. If part 1 of a computation must be done before part 2, having two processors will not help to expedite the response.

In our example, we are ambiguous about whether the two servers can be used concurrently. If, prior to sending a service request to the second server, the computations can be accomplished without the results of the request to the first server, the two requests can be serviced concurrently. However, if, prior to the request to the second server, the computations depend on the results of the request to the first server, the two requests cannot be serviced concurrently.

2. Overhead in using services, message marshaling, and transferring control introduces extra execution time, which can be reduced by caching the service location, restricting message formats, and using efficient transfer-control mechanisms.

   For the first request in our example, the name service must be invoked. However, the handle for these services can be either reacquired for every request or maintained within the client until the server is relocated. The method chosen will affect latency.

   Message marshaling can also routinely consume significant execution time depending on message formats, compiler representations of data, and other factors. Message construction rules can reduce marshaling overhead.

   Also, there is usually OS overhead associated with starting and stopping computations. When an event's response is split over multiple resources (such as multiple processors), the transfer of control from the first resource to the second is not instantaneous—there is a transfer time associated with it. Sometimes this transfer time is simply the overhead associated with method invocation; in other cases, it is the result of carrying a complex protocol over a network. Transfer time is another significant contributor to latency. Using Remote Procedure Invocation (RPI), for example, would involve greater control-transfer overhead than packaging the procedures into the same process and using a direct procedure call would.

3. Algorithms, especially in places with high demand, have a strong impact on the execution time.

   The request for a service is eventually executed by a server. This execution can be done in either an efficient way or a time-consuming way. If the request is to sort a list, different sort algorithms can be used, each of them influencing the execution time to some degree.

4. Not taking resource arbitration criteria, such as deadlines or semantic importance, into consideration can lead to increased latency for events that can afford it the least. This is exacerbated by variability in resource demands (such as message-processing time) that can lead to long queues. Using priority queues in which the priority is based on timing requirements and/or semantic importance can help.

   Many message protocols can use a single buffer for all messages and remove them from the queue in a FIFO manner. Highly important messages or short messages with tight deadlines can be stuck behind less important messages or large messages with distant deadlines.

4.4 Performance Tactics

In general, latency is affected by the level and nature of the demand for resources. This includes event-arrival rates, the execution time resulting from event arrivals, and the variability level of each. Latency is also affected by the rules for choosing between conflicting demands for a resource. The inability to use a resource, either because it is being used to process other events or is physically (e.g., due to failure) or logically (e.g., due to inadequate resource-allocation policies) unavailable, also affects latency. Therefore, we divide performance tactics into three high-level groups:

1. tactics for managing resource demand—These tactics control waiting times and transfer times by controlling the demand for resources.
2. tactics for arbitrating between conflicting demands—These tactics control preemption and waiting times when there are competing requests and take semantic importance or urgency (deadlines) into consideration when there is contention for shared resources.
3. tactics for managing multiple resources—These tactics enable multiple resources to be used efficiently to ensure that available resources are used when they are needed, thereby controlling the waiting time for them.

These tactics are described in Sections 4.4.1 through 4.4.3 and analyzed in Section 4.5. Keep in mind that our goal is not to create a taxonomy of tactics and that some tactics may fit into more than one category.

4.4.1 Tactics for Managing Demand

Event streams are the source of resource demand. Two stream traits characterize demand: the time between events in an event stream (i.e., the arrival rate) and how much of a resource is consumed by each request (a.k.a., execution or service time). The variances in arrival rates and execution times also affect latency.

1. **Manage the event rate.** Tactics in this category control how often events are generated. For example, reducing the sampling frequency at which environmental variables are monitored or using a name server once and caching the result (rather than using a name server every time a request has to be sent directly to a server) changes the rate at which events are produced. Sometimes this is possible if the system was over-engineered. Other times, doing interpolations between lower rate measures is good enough. Throttling is another example that involves slowing down the rate at which messages are produced to accommodate limitations downstream in the ability to consume messages and avoid data loss.
2. **Control the frequency of sampling external events.** If there is no direct control over the rate of externally generated events, queued events can be sampled at a lower frequency, possibly resulting in the loss of requests. Also, multiple events can be aggregated before processing; effectively reducing the overhead of setting up the context before the event can be processed. The context setting is done once for the aggregated events rather than once for each event.

3. **Reduce the computational overhead.** If OSs and middleware, for example, were infinitely fast, an important source of overhead would be eliminated. However, computational work usually requires OS and middleware services to manage process interaction, communications, and the like. Therefore, adjusting the assignment of responsibilities to processes influences the need for communication between those processes. For example, using Remote Method Invocation (RMI) for requesting a service from a process incurs more overhead than using a Java™ class method within the same process. Assigning computation to processes to reduce context-switching overhead is another example.

   Section 4.5 describes how latency grows as a consequence of variability. The reason this occurs is that bursts of events or long execution times allow queues to build up. Therefore, bounding variability is a tactic for controlling latency. Execution-time variability and/or arrival-rate variability can be bounded.

4. **Bound execution times.** This means placing a limit on how much execution time is used to respond to an event. Sometimes this makes sense and other times it doesn't. For a data-dependent, iterative algorithm, bounding execution time can be accomplished by limiting the number of iterations. However, this might imply trading off accuracy for improved latency.

5. **Bound queue sizes.** This tactic directly controls the maximum number of queued arrivals and effectively bounds arrival rates, but the consequence can be event loss. When this happens, you need to calculate (or estimate) the probability of losing events based on the chosen queue length and determine if that loss rate is acceptable. You also need to determine if the lost events are newly arriving events or events that have been queued for a while. Sampling stochastic arrivals periodically at intervals equal to the arrival rate transforms a stochastic stream into a periodic stream, again with the possibility of some data loss.

6. **Increase the computational efficiency of algorithms.** One step in processing an event or a message is applying an algorithm to perform the processing. Improving the algorithms used in critical areas reduces the demand for processor time and therefore has the effect of improving latency. Sometimes reducing the precision with which a calculation must be performed reduces the computational requirement. Additional overhead also results from catering to other attributes. The use of intermediaries (so important for modifiability) increases the computational load.

7. **Control the demand for resources.** When an algorithm requires the use of other resources, such as disks, its efficiency also depends on how those resources are used. For example, reading the complete file from a disk at the beginning of the computation is more efficient than reading the file line by line during the computation.

4.4.2 Tactics for Arbitrating Demand

When multiple streams make demands on the same (logical or physical) resource, tactics are needed for managing the shared resource. One of the main considerations for arbitration[1] is the criterion used for selecting which of a set of competing streams should use the resource.

™ Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

1 We use the terms *arbitration* and *scheduling* interchangeably.

Time-based, semantics-based, and fairness-based criteria are all considered. The fact that sometimes the resource is non-preemptable must also be considered. If such decisions are to be made at runtime, support for multiprocessing is helpful.

1. **Increase the logical concurrency.** Mechanisms for achieving logical concurrency (such as processes and threads) allow the separation of concerns associated with processing event streams from the interleaving of such processing on a physical resource. Processes and threads allow the processing of one stream to preempt the processing of another.
2. **Determine the appropriate scheduling policy.** Many tactics are focused on algorithms to assign processor time to processes. The appropriate scheduling algorithm strongly depends on the system's goals. The scheduling policy of a real-time system that has to obey hard deadlines is very different from the scheduling policy of a multi-user information system, where all users should be treated equally. Scheduling tactics include

   - offline scheduling. A schedule can be constructed offline using assumptions about the time to be taken by each stream. This tactic assigns to each stream a particular period of time and a priority within the schedule. The "logical processes" for each stream are woven together manually or mechanically. In effect, all scheduling decisions are made *a priori*, offline. During execution, the scheduler assigns the processor to the streams in priority order for a fixed period of time. This tactic (often called a cyclic executive) results in a schedule in which the latency for each stream can be predicted based on its order in the schedule and the period each stream is assigned. However, this tactic introduces the resource dependency discussed in Section 3.4.3 and leads to many perturbations when a modification occurs.
   - time-based scheduling. Informally, a "good" prioritization strategy takes into account the timing requirements associated with the stream (or even different arrivals within the same stream). Therefore, stringent timing requirements, such as relatively tight deadlines, should have priority over arrivals with relatively distant deadlines. This is reflected in three prioritization strategies:
     - rate monotonic (RM) strategy—a static priority assignment (i.e., the priority stays the same for all arrivals in the stream) for periodic streams that accords higher priorities to streams with shorter periods than to those with longer ones
     - deadline monotonic (DM) strategy—a static priority assignment usually for periodic streams that accords higher priorities to streams with tighter deadlines than to those with more distant deadlines.
     - earliest deadline first (EDF) strategy—a dynamic priority assignment (that is, the priority can change for each arrival in the stream), usually for periodic streams, that accords higher priorities to streams with tighter deadlines than to those with more distant ones
   - semantic-importance-based scheduling. Semantic importance is another scheduling criterion. The importance of different functions can vary by system mode. Therefore, the system must focus its resources on the appropriate work at the appropriate times. For example, in cases where overload occurs and not all streams can meet their deadlines, the goal is to shed the least semantically important load. The aforementioned strategies might not perform adequately in such situations, even though they are designed from the point of scheduling, because they do not consider semantic importance. There are techniques, such as the Quality of Service (QoS) Resource Allocation Method (QRAM) [Lee 98], that can be used in concert

with time-based scheduling to both maximize schedulable utilization and ensure that the most important work gets done. The QRAM introduces the notion of system utility and provides a method for maximizing it as a function of resource utilization.

- aperiodic servers. Another tactic for increasing the overall utility of a system focuses on computational work that is normally relegated to background processing. While it is important for events with hard deadlines to complete their processing *by* those deadlines, in many cases, there is no value (or utility) associated with completing them significantly *before* those deadlines. However, reducing the average latency associated with streams that might receive only background processing might be of value. In such a case, this extra "high-priority" processing time can be used to decrease the average latency of the so-called background processes. Aperiodic servers carefully "steal high-priority time" in a way that does not affect the schedulability of processes with stringent timing requirements. At the same time, these servers improve the average latency of processes with important but softer timing requirements.
- fairness-based scheduling. Several scheduling strategies that do not account for deadlines are FIFO scheduling and processor sharing. They do not take urgency or importance in mind, but rather use fairness as the scheduling criterion. As a result, they can be problematic in performance-oriented systems such as real-time systems.

3. **Use synchronization protocols.** When more than one thread of concurrency needs to access a shared resource, such as a queue or a data repository in a mutually exclusive manner, mechanisms such as locks and semaphores are commonly used. The shared resource is locked by one thread, making it inaccessible to other threads until the thread holding the lock is finished with the resource and unlocks it. Synchronization protocols govern the rules for "scheduling" the resource. Scheduling determines which waiting thread is allowed to lock the resource next and the execution properties of the resource-locking thread [Rajkumar 91].

   Synchronization protocols have an important impact on latency, since they determine how long the threads have to wait to acquire the lock. Section 4.5 illustrates how waiting times that might be arbitrarily long can be managed properly and bounded appropriately.

4.4.3 Tactics for Managing Multiple Resources

Exploiting the power of multiple resources to achieve performance goals introduces additional challenges to those managing single resources. Introducing additional processors for example, offers opportunities to reduce latency by exploiting physical concurrency. However, physical concurrency brings with it the need to make resource-allocation decisions, such as which processor(s) should be used to process which event streams. When processing is physically distributed, the results computed on one processor are often needed by other processors, introducing the need for communication and synchronization. Making local copies can help reduce the overhead associated with moving data around.

1. **Increase the physical concurrency.** A tactic for reducing latency is to increase the number of available resources. Additional processors, additional memory, and faster

networks all have the potential for reducing latency by introducing the possibility of parallelism. When there are increased resources, replicated processes can exploit them. Clients in a Client/Server pattern are replicates of a computation. The purpose is to reduce the contention that would occur if all computations occurred on a central server. Performing I/O offers another common opportunity to exploit physical concurrency by performing computations while I/O is occurring.

2. **Balance resource allocation.** Appropriately allocating the load between multiple resources is important if physical concurrency is to be exploited maximally. One strategy for this is balancing the load. However, it can be shown that load balancing doesn't always result in the best latency. Another criterion for allocating processes to processors is to minimize communication overhead.
3. **Increase the locality of data.** A tactic for reducing latency is to maintain multiple local copies of data. Caching replicates data on different speed repositories or on separate repositories to reduce contention. Since the data being cached is usually intended to be replicates of the same data, keeping the copies consistent and synchronized becomes a responsibility that the system must assume.

4.5 Performance Analysis

This section offers several sample performance analyses to give you a sense of the level of detail we think is appropriate. This is an issue with which we are still wrestling. If the analysis is too qualitative or not detailed enough, it might be generally applicable but not very insightful. Our intent is not to write a treatise on scheduling and queuing theory; this is our first attempt at discovering an appropriate and hopefully powerful middle ground.

Our strategy for discovering this middle ground is to find the points of leverage in the analysis. For performance, these are the independent parameters of the performance model that will have a dramatic effect on latency. These leverage or sensitivity points are the motivation for the performance tactics and the focal points for our analysis.

In this report, we focus on analyzing for latency. *Latency* is how long it takes from the time a request for service is made until the request is fulfilled. Latency is influenced heavily by each stream's average interarrival time and average service time (a.k.a. execution time). Frequently, there is also some notion of a deadline; achieving latency *before* the deadline is usually more desirable then achieving latency *after* the deadline.

Next, we concentrate on some of the factors that lead to more or less latency. In Section 4.5.1, we discuss how variability in event-arrival rates and execution times influences latency. In Section 4.5.2, we discuss the effects that multiple streams have on each other. Managing multiple resources is not discussed, because it is beyond the scope of this report.[2]

[2] We hope to include information on managing multiple resources in a future report.

4.5.1 Managing Demand

To gain insight into the effects of demand on latency, we focus on a relatively simple situation. Consider a single event stream that requires the use of a single resource for its computational work. The stream's events are processed using a first-come, first-served (FCFS) or sometimes FIFO scheduling policy. You can think of a stream of events as a sequence of messages or clock interrupts.

For the analysis and architectural tactics considered in this section, we assume that specific information about events—such as whether they are of the same type, have different deadlines, have different semantic importance, or have associated priorities—is either ignored or unimportant.

4.5.1.1 General Reasoning About the Effects of Demand

Before we start, here are some notational conventions used in this report:

- *T*—denotes latency, the time from when an event occurs until its processing is complete. This includes queuing time and service time.
- *Q*—denotes the time that an event spends in a queue waiting for service
- *S*—denotes the average service time[3]
- *E[T]*, *E[Q]*, and *E[S]*—denote the expected value of (or mean) latency, queuing time, and service time, respectively

Given that there is only a single stream, which is served in FIFO order, each event can be processed only after all prior arrivals have completed. In this case, the average latency is equal to the average amount of time the event request spends in the queue:

$$E[T] = E[Q] + E[S]$$

Therefore, a key question is how many prior arrivals can be in the service queue when a new arrival occurs? The answer depends on the relationship between the average interarrival time and average service time (*E[S]*), and on the variability of each. For example, for a periodic stream, if the worst-case execution time is less than the length of the period, the queue will always be empty when a new service request arrives. In this case, the latency is simply the service time. On the other hand, if event interarrival intervals are not regular or if an event's service time can sometimes be extremely long, many new event arrivals can occur while an event is being serviced. This will result in a buildup of requests in the queue.

From this high-level reasoning, it is evident that three of the tactics for managing demand—(1) manage the event rate, (2) increase the computational efficiency of algorithms, and (3) bound queue sizes—have an influence on latency. Managing the event rate affects the inter-

[3] We use the terms *service time* and *execution time* interchangeably.

arrival times. Increasing computational efficiency decreases the service time, and bounding queue sizes decreases the amount of time that a request spends in a queue.

4.5.1.2 Tactics for Managing Demand

While simplistic, the FIFO queue has a very significant influence on the performance of systems. It is simple to implement and pervasive and, yet at times, can be inimical to achieving performance goals. It also sheds light on the impact of execution time, event-arrival rates, and variability on latency.

In the simplest case shown in Figure 5, a single process with a single queue serves a single stream of messages. We assume that it takes a negligible amount of time to get data into and out of the queue. Because the queues are FIFO, scheduling is simple. The single process monopolizes the whole CPU and runs continuously until it's completed.

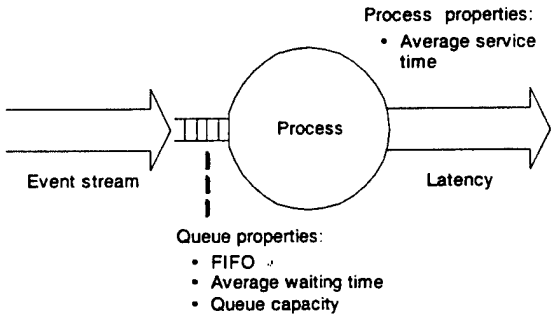


Figure 5: A Single Process with a Single Queue that Serves a Single Stream of Messages

Latency is composed of waiting time and service time. The waiting time is also composed of two parts: (1) the amount of work that is in the queue and (2) the work that remains to be completed for the event being serviced. The following formula accounts for both parts:

$$E[Q] = (U / (1-U)) * E[R].$$ where

U is the average utilization— the average amount of time the server is busy. *U* is equal to the average service time divided by the average interarrival time (1-*U*). *E[R]* is the expected amount of remaining work for the event being processed when a new event arrives. The average latency, *E[T]*, equals the average time in the queue, *E[Q]*, plus the time in the server, *E[S]*:

$$E[T] = (U / (1-U)) * E[R] + E[S]$$

The nature of *E[R]* depends on the specifics of the execution time. Table 2 covers several important cases with different assumptions about the execution time.

Case	E[R]	E[T] = (U/(1-U)) * E[R] + E[S]	Comments
Poisson arrivals and constant execution times	E[S]/2	(U/1-U) * (E[S] / 2) + E[S]	On the average, a new arrival will see that half the work has already been done. Therefore, E[R] is half the execution time.
Poisson arrivals and exponential execution times[4]	E[S]	(U/1-U) * E[S] + E[S]	Since exponentials are "memoryless" (i.e., they don't "remember" completed work), E[R] is the entire average execution time.
Poisson arrivals and general stochastic execution times with mean E[S]	$E[S^2]/2E[S]$	(U/1-U) * $E[S^2]$ / 2E[S] + E[S]	E[R] accounts for the possibility of significant variability in execution time.

Table 2: Expected Remaining Work and Expected Latency for Different Assumptions About Arrival and Execution Distributions

In the most general case, $E[R] = E[S^2] / 2E[S]$, we can rewrite E[T] as follows:

$$E[T] = (U / (1-U)) (C^2 + 1) (E[S] / 2) + E[S], \text{ where } C = \text{Stddev}(S) / E[S]$$

First, note that the formula is true when event arrivals follow a Poisson distribution. This is a common assumption made in queuing theory. The following two conditions hold for Poisson arrivals and offer some intuition for whether a distribution is Poisson:

- independent increments—The number of arrivals that occur in disjoint intervals of time are independent.
- stationary increments—The number of arrivals that occur during an interval of time depend only on the length of the interval; they do not depend on the absolute time at which the interval begins.

The above formula reveals why controlling the arrival rate and execution time, and bounding variability are important for controlling latency. The execution time contributes to latency directly through *E[S]* and indirectly through the utilization term, *U*. The arrival rate contributes to latency through *U* in the factor *U / (1-U)*. Recall that utilization is the average service time over the average interarrival interval, which is the reciprocal of the average arrival rate. And variability contributes to latency through *C* in the factor (*C*2 + 1).

[4] This case is often denoted in queuing theory as M/M/1. The Ms stand for memoryless, which is a property of the exponential distribution. Poisson arrivals mean the number of arrivals that occur in any interval of time and are described using the Poisson probability distribution. Poisson arrivals imply exponential interarrival times and thus the first letter M. Exponential service times are denoted by the second letter M. And the number 1 means that there is a single server.

Utilization, *U*, has a significant effect on average latency through the term *U* / (1-*U*). To illustrate, assume that *E[S]* is 2 seconds and the standard deviation is 0. When the utilization is .1, the average latency, *E[T]*, is a little greater than 2.1, where most of it is attributable to *E[S]*. On the other hand, if *U*=.9, the average latency is 11, where most it is attributable to the high level of utilization. Figure 6 shows how average latency varies as a function of utilization. This figure was derived by fixing *E[R]* and *E[S]* and varying the utilization. Controlling the execution time and arrival rate has a direct impact on utilization and thus is important for controlling latency.

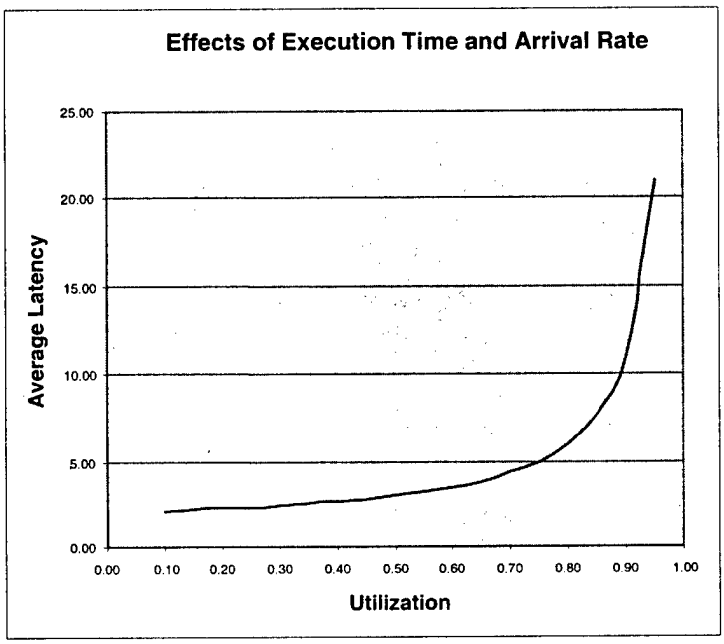


Figure 6: Average Latency as a Function of Utilization for Assumed E[R] and E[S]

Now consider the variability in execution time. This is captured by the parameter *C* in the equation

$$E[T] = (U / (1-U)) * (C^2 + 1) * (E[S] / 2) + E[S], \text{ where } C = Stddev(S) / E[S]$$

C is a normalized standard deviation—the standard deviation of the execution time divided by the mean execution time. To illustrate, assume that *E[S]* is 2 seconds and the utilization is .5. Then consider the following cases:

- If *C*=0, *E[T]* = 3 seconds. (This is known as M/D/1 in queuing theory.)
- If *C*=1, *E[T]* = 4 seconds. (This is known as M/M/1 in queuing theory.)
- If *C*=2, *E[T]* = 7 seconds, a significant difference.
- If *C*=5, *E[T]* = 28 seconds, demonstrating the drastic effect of variability.

Figure 7 shows how average latency varies as a function of C.

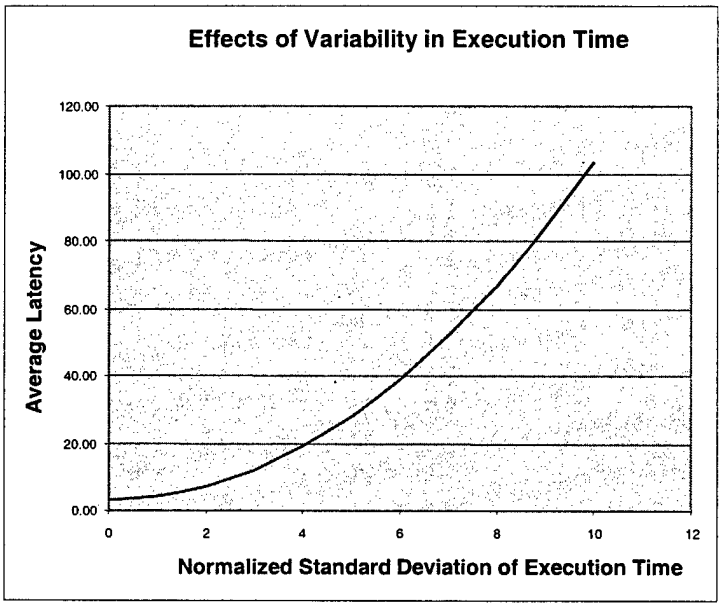


Figure 7: Average Latency as a Function of C

Notice that even at moderate levels of utilization when the mean service time is significantly less than the mean interarrival time, variability can have a significant impact on latency.

Next let's take a quick look at variability in arrival rates. Take the M/D/1 case, which has a deterministic execution time, and alter the arrival rate so that two arrivals must be separated by a minimum interarrival interval. This reduction in variability can further reduce latency. For example, if the minimum interarrival interval is 4 (consistent with a maximum utilization of .5), the latency will be equal to the execution time, which in this case is 2 seconds. Thus, managing the event rate is a key tactic for controlling latency.

4.5.1.3 Observations About Managing Demand

Multiple streams converge into a FIFO queue. A minor but common variant of the FIFO queue is when multiple streams of different types of messages converge on the same queue, but are treated as a single stream served by a single process. This is shown in Figure 8. Clearly, the streams affect one another, but their timing requirements (and semantic importance) are assumed to be the same (or at least very similar). In this case, multiple streams are treated as a single stream. The single process monopolizes the whole CPU and runs until it's completed.

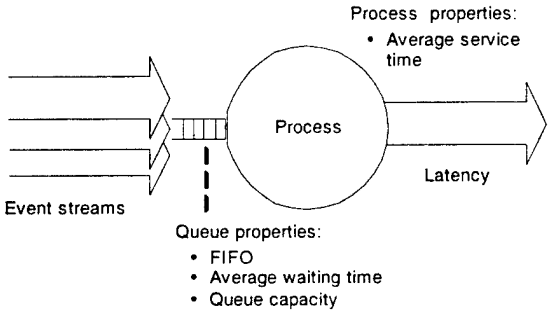


Figure 8: Multiple Streams Being Treated as a Single One Served by a Single Process

Do you really have a single stream? It may appear as though there is a single stream, and yet, upon closer examination, you may discover that several message types with different arrival rates and different timing requirements are converging on the same server. For example, a collection of periodic streams that are handled in FIFO order by a single process could look like a single stochastic stream. However, this would not be a suitable design alternative if each stream had its own timing requirements.

There may be many sources of execution time. When considering execution time, you must not only consider the time associated with executing some specific function, but also any overhead associated with the OS, middleware, communications, and so forth. This is the focus of the tactic for reducing computational overhead. The analysis above shows how overhead can affect utilization, which, in turn, can affect latency.

Where are the queues? Many times, queues are readily apparent and used by an application to hold pending requests for processing, for example, a queue of pending messages. Other times, a pool of threads may be available to handle incoming events. When an event occurs, it is allocated a thread, and the processing is carried out within that thread. In this case, the queue is less apparent; the thread dispatcher has a queue of pending threads.

FIFO is simple, common, and sometimes problematic. Using a single-stream, FIFO queue is a relatively simple situation. While it might seem unrealistically simple, FIFO queues show up quite a bit in practice. They treat all arrivals as equals; however, be leery of FIFO queues when all arrivals are not equal.

4.5.2 Arbitration

To gain insight into the effects of arbitration, we focus primarily on priority scheduling. Since variability was a primary consideration in the previous section, we don't focus on it

here. Rather, we focus on the effects that one stream has on another. In particular, we consider multiple streams of events, each with bounded interarrival and execution times.

4.5.2.1 General Reasoning About the Effects of Arbitration

In the single-stream case, the three main factors contributing to latency were

1. the mean and level of variability in execution times
2. the mean and level of variability in interarrival times
3. the utilization level

These continue to be factors. However, in this section, we focus on the effects of prioritization strategies and scheduling in catering to the differing needs of different streams.

Whereas for FIFO queuing the waiting time was determined by how many queued requests a new arrival would face on average, when priority queuing is used, those already in the queue do not have a secured position. New arrivals can jump ahead of them based upon their priority. Priority scheduling is therefore a powerful mechanism to lower the latency of those requests that possess relatively high priorities.

However, some component properties work against priority scheduling. If an event in a high-priority stream arrives only to find that it cannot preempt the processing associated with a lower priority stream, some of the power of its high priority is diminished. Therefore, you can view preemption as being supportive of priority scheduling and non-preemption as being unsupportive of it.

For priority scheduling, it is useful to think about the following effects on a job's latency:

- preemption—the effect of higher priority jobs
- execution time—the effect of the jobs' execution times
- blocking time—the effect of lower priority jobs

As in the previous section, execution time clearly affects latency. Moreover, just as a burst of arrivals to a FIFO queue can dramatically affect latency, a burst of higher priority arrivals from higher priority streams can also dramatically affect latency. For example, even if all streams are periodic (i.e., each stream is regular and not prone to having bursts of arrivals), every now and then, the arrivals from each stream might occur at the same time. The alignment of high-priority streams will dramatically affect the latency of lower priority streams. This instant, known as the critical instant, is important for determining worst-case latency.

Additionally, tasks with lower priority assignments can block the execution of tasks with higher priority assignments. This can happen when two tasks share a resource that requires

mutually exclusive access (e.g., via binary semaphore or a monitor) or when a task executes a non-preemptable section.

The three tactics that are most obviously relevant to the analysis covered in this section are (1) increase the logical concurrency, (2) determine the appropriate scheduling strategy, and (3) use synchronization protocols.

We use the priority-based preemptive scheduling of periodic processes as our starting point. While this is a very specific case, it is a good starting point for understanding how multiple streams interact and how that interaction affects latency, and for motivating the use of time-based scheduling tactics.

The formula for latency of process i is shown below. S_i denotes the execution time of process i, P_i denotes the period of process i, and B_i denotes the blocking time incurred by process i. The worst-case latency for process i, assuming that processes 1 through i-1 are of higher priority, can be found by iterating the following formula until it converges (i.e., the value of T_n remains the same for two consecutive iterations). Worst-case latency is often of interest to developers of real-time systems, where meeting deadlines is critical.

$$T_{n+1} = \sum_{j=1}^{i-1} \left\lceil \frac{T_n}{P_j} \right\rceil S_j + S_i + B_i$$

The first term in the equation above reflects the use of a priority-based, preemptive scheduling policy and computes the number of times that higher priority processes can preempt process i in a window of time that starts at time 0 and extends to time T_n. With each iteration, the formula accumulates the execution time associated with each of these preemptions, adds in the execution time of process i (S_i), and adds in the blocking time (B_i). As stated above, the blocking terms capture the effects of lower priority processes. (We enumerate several sources of blocking below.)

Start by assigning T_1 a value of S_i and iterate until T_n equals T_{n+1}. The result is a determination of the worst-case latency for process i, or more precisely, a bound on the worst-case latency. If the iterations do not converge or converge beyond the process's deadline, there may be potential timing problems. The lack of convergence signals an unbounded latency.

In a concise manner, the above equation motivates several tactics. First, it illustrates the sensitivity of latency to higher priority processes. Notice that the formula does not embody any specific prioritization strategy. It simply accounts for all of the processes that happen to have a higher priority than process i. However, you can easily imagine a process with a very long period, a very distant deadline, and a very long execution time that was accorded a very high priority. Such a process might meet its deadline, while the other processes don't meet theirs.

This observation offers some intuition for the tactic of using time-based scheduling. The RM and DM scheduling subtactics assign priorities based on periods and deadlines, respectively. They cater explicitly to the different timing requirements of different streams.

The above equation also highlights blocking time as a contributor to latency. Synchronization required by mutually exclusive access to a shared resource is one source of blocking. A classical synchronization analogue to the above example of priority assignment follows.

Assume that process 1 has a high priority, process 2 has a medium priority, and process 3 has a low priority. While the low-priority process has exclusive access to some shared resource (the section of a process that has exclusion access to a resource is known as a critical section), the high-priority process will have to wait if it also needs the resource. The amount of time that it waits is the blocking time. This could happen easily if the high-priority process preempts the low-priority process, while the latter is using the shared resource. The medium-priority process could further exacerbate the situation by preempting the critical section and causing an even longer delay for the high-priority process. For other medium-priority processes, the blocking time could increase immensely.

In this case, the key insight is that medium-priority processes were able to preempt the critical section. This motivates the tactic of using synchronization protocols, such as the priority ceiling and priority inheritance protocols. Such protocols would not allow the critical section to be preempted by a medium-priority process.

4.5.2.2 Observations About Arbitration

Identifying sources of blocking. In general, blocking occurs whenever an architectural design permits a low-priority process to execute when a higher priority process is also ready to execute. In some cases, blocking is unavoidable. In others, synchronization protocols can be used to reduce blocking. Several common sources of blocking include

- critical section. As discussed above, a critical section is a source of blocking. An improper execution priority during the critical section can result in an unnecessarily large amount of blocking. The key to circumventing this is to ensure that medium-priority processes do not have an opportunity to preempt the critical section that is blocking the high-priority process. One such prevention technique (in a uniprocessor setting) is to set the priority of the critical section to be slightly higher than that of the shared resource's highest priority client.
- deadlock. Deadlock is an extreme form of blocking in which processing comes to a halt. It occurs when two or more processes need mutually exclusive access to two or more of the same resources. Deadlock is discussed in most books on OSs.
- FIFO queue. A FIFO queue is another common source of blocking. If a high-priority process is stuck in a FIFO queue behind a lower priority process, the blocking time can be arbitrarily long.

- non-preemptable section. Sections of low-priority processes that are non-preemptable can delay a higher priority process. This is because the latter is prevented from preempting if it is time for it to execute and the low-priority process is in its non-preemptable section.
- interrupt. Strictly speaking, interrupts are not a source of blocking, but rather a source of preemption. However, often a low-priority thread is initiated by an interrupt. Since the interrupt is executing on behalf of the low-priority process, it can be viewed as a source of blocking to other higher priority processes.
- threads and processes. Some OSs support threads, which are lightweight units of concurrency that execute within a single, shared address space (whereas each process executes in its own address space). Sometimes in this situation, a two-level scheduler is used. That is, first, processes are scheduled, and then threads within those processes are scheduled. A thread's high priority can be virtually ineffective if the thread resides in a process that has been assigned a relatively low priority.
- pipelines. The equation above models collections of periodic processes. What if there is a more complicated topology such that several processes have precedence relations? For example, if the initial process in a sequence of processes is initiated by a clock interrupt or a message arrival, while the next process in the sequence is initiated at the completion of the first one and so on, how do we reason about this situation? Here are some example rules of thumb to help think about this.

  For example, let's say that the whole pipeline can finish its processing within the period and that all processes have the same priority. From a performance point of view then, this situation is equivalent to having a single process whose execution time is the sum of the processes' execution times and whose priority is equal to the processes' common priority.

  Next let's say that the whole pipeline can finish its processing within the period and that processes can have different priorities. For the latency analysis of this specific pipeline, you can transform the pipeline into a single process whose execution time is the sum of the processes' execution times and whose priority is equal to that of the lowest priority process in the pipeline. You can think of this as the "pipeline is only as good as its weakest link" (i.e., its lowest priority). The latency of the "transformed architecture" is no better than the latency of the original.

  The above rules also work if the processing of the pipeline can continue past the end of the period, but you can ensure that the processing of earlier messages always has a higher priority than the processing of later messages.

  If the processing of the pipeline can continue past the end of the period and messages that arrive later can be processed before earlier messages have completely traversed the pipeline, this preemptive effect of later messages on earlier messages must be accounted for.
- non-periodic processes. When processes are not periodic, there is a minimum interarrival interval, and the difference between the minimum and maximum interarrival intervals is not too large, the minimum interarrival interval can be viewed as a period, and the above analysis can be performed.

5 Using Tactics to Analyze Patterns

In this section, we turn to our detailed example and use our tactics to explain and analyze patterns that are used in the Java Adaptive Web Server (JAWS):[5] specifically the Half-Sync/Half-Async and Wrapper Façade patterns [Buschmann 00]. We analyze both patterns for performance and modifiability.

In Section 5.1, we describe both patterns and their components and how those components interact to provide services. In Section 5.2, we analyze the patterns for modifiability when they are used in the JAWS. Lastly, in Section 5.3, we analyze the patterns for performance.

5.1 The Constructive Aspects of the Patterns

5.1.1 Half-Sync/Half-Async Pattern

The Half-Sync/Half-Async architectural pattern decouples asynchronous and synchronous service processing in concurrent systems to simplify programming without unduly reducing performance. As shown in Figure 9, this pattern introduces two intercommunicating layers for service processing: asynchronous and synchronous [Buschmann 00].

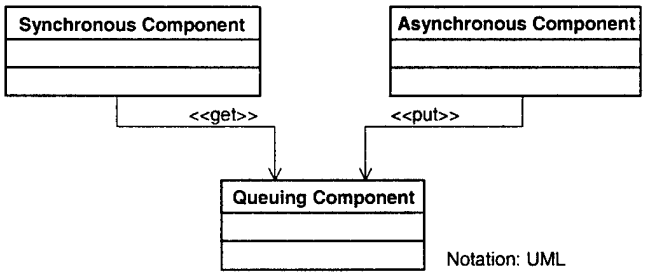


Figure 9: Interaction Among Components in the Half-Sync/Half-Async Pattern

Writing applications that require input over a network can be simplified by providing synchronous calls, such as a `get(data)` call that waits if no data is present and returns as soon as data is available. Synchronous calls are easier to handle than programming schemas, such as polling with waiting, or setting up events with waiting until the event occurs. Using a synchronous call also means giving up control over concurrency. Making a `get` call defi-

[5] JAWS is a Web server and a framework from which other types of servers can be built.

nitely suspends the activity, while the other methods allow doing something else while waiting for input. The possible performance issue from blocking that comes with using synchronous calls can be alleviated by using multiple threads.

On the network side, messages will arrive asynchronously. Using synchronous and asynchronous programming styles together requires a synchronization, which is done by the queuing component. This component suspends and resumes synchronous `get` requests, provides a `put` service to allow placing data into the queue, and handles the serialization of concurrent requests to the queue.

The typical flow of events is that an application calls the `get` service of the queuing component, which returns the next data item from the queue. During this time, the queuing component is locked. When the queue is empty, the `get` request is suspended and the lock is released. As soon as a data item is placed into the queue by an asynchronous component, the suspended `get` request resumes with the data item that was just placed into the queue. When a `get` or `put` request arrives at the queuing component while an earlier request is still active, the request is blocked until the earlier request is finished.

In the JAWS, the Half-Sync/Half-Async pattern is used to provide a synchronous interface that Web server applications can call to receive hypertext transfer protocol (HTTP) messages. These messages are sent by a client (e.g., Web browser) packed into Transmission Control Protocol/Internet Protocol (TCP/IP) messages via a network and received asynchronously by HTTP handlers. These handlers, in turn, unpack the HTTP messages and place them into a queue for the Web server applications.

5.1.2 Wrapper Façade Pattern

The Wrapper Façade design pattern, shown in Figure 10, encapsulates the functions and data provided by existing non-object-oriented application program interfaces (APIs) within more concise, robust, portable, maintainable, and cohesive object-oriented call interfaces [Buschmann 00].

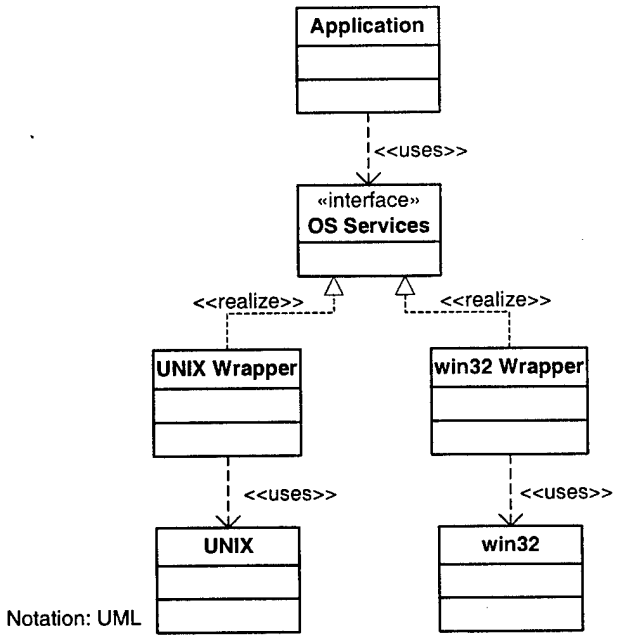


Figure 10: Wrapper Façade Pattern

The Wrapper Façade pattern defines an interface to applications that abstracts from implementation details of the service providers. The pattern's components do the necessary adaptations from the services offered by the specific service provider to the services published to the applications.

The structure depicted in Figure 10 is simplified. Depending on how it is used in a system, the pattern may involve several interfaces and components. Moreover, since the purpose of the pattern is to translate from an object-oriented interface to a non-object-oriented interface and vice versa, multiple instances of the pattern's components are possible. For example, if a service of the pattern is to provide a thread interface to applications, it is likely that instead of managing "thread handles" delivered by the underlying OS, thread objects will be used to encapsulate the necessary thread-managing tasks.

5.2 Analyzing Patterns for Modifiability

5.2.1 Analyzing the Half-Sync/Half-Async Pattern for Modifiability

In the JAWS, HTTP messages for several active applications are received over a network by an HTTP handler, as shown in Figure 11. A received HTTP message is processed by one of the Web server applications. These applications are implemented to use a synchronous `get` service, which suspends the application until the required HTTP message arrives. This re-

quires the exchange of HTTP messages between the HTTP handler and the Web server applications to be done indirectly via a queue implemented in the request queue component.

Therefore, the Half-Sync/Half-Async pattern uses the tactic of breaking the dependency chain to remove the service dependencies between the HTTP handler and the Web server. This enables the two components to implement different service-processing strategies (synchronous and asynchronous). The request queue component acts as the intermediary.

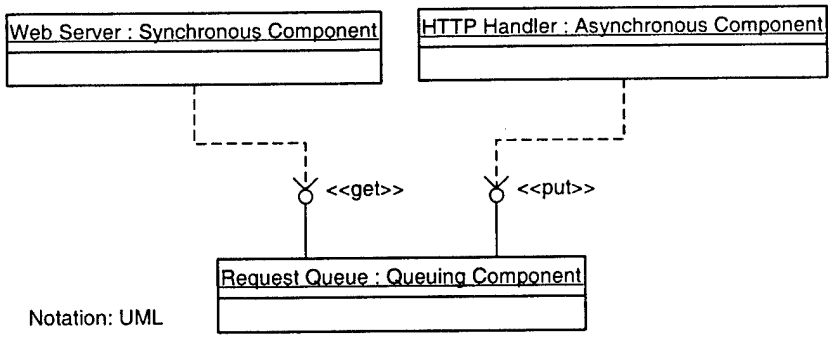


Figure 11: Half-Sync/Half-Async Pattern Applied in the JAWS

The results of analyzing the dependencies that occur when the Half-Sync/Half-Async pattern is applied are described below.

5.2.1.1 Data-Syntax and -Semantic Dependencies

The HTTP handler exchanges HTTP messages with the Web servers. Therefore, those components have a mutual dependency on the data semantics (through HTTP) of each other. The data exchange is done indirectly via the request queue component. This component doesn't know about the semantics of the exchanged messages but must know their syntax. The purpose of the request queue as applied in the JAWS is not to translate one syntax into another, but instead to merely hold the messages until they are required. This means that all three components have to understand the syntax of the HTTP messages, thereby adding data-syntax dependencies to all of them.

5.2.1.2 Service-Syntax and -Semantic Dependencies

The HTTP handler and the Web servers do not use services directly from each other. Instead, they communicate by exchanging data (HTTP messages) via the request queue. Therefore, both the HTTP handler and the Web server must understand the syntax and semantics of that queue's services, but they do not have service dependencies on each other.

5.2.1.3 Identity, Existence, and Runtime Location Dependencies

The request queue's services are used by direct call (method invocation). This requires the HTTP handler and the Web server to have knowledge of the request queue's identity and runtime location. The use of method invocation also requires that the queue exists.

There is also a mutual existence dependency between the HTTP handler and the Web server that is not quite that obvious. Although the two components don't communicate directly with each other, they are still dependent on one of them delivering HTTP messages and the other processing them. If the HTTP handler didn't exist, no message would be delivered, and the Web server would wait infinitely and do nothing. If the Web server didn't exist, the request queue would fill up, and the HTTP handler would no longer be able to function.

5.2.1.4 Data- and Control-Sequencing Dependencies

The request queue in this example handles the HTTP messages in a FIFO manner, but the fact that the data items in the queue are HTTP messages makes the pattern independent from any queuing/dequeuing mechanism. This is because every HTTP request from a client (e.g., Web browser) is translated into exactly one HTTP message. Even processing the latest message first would not affect the correct functionality of the applications (although it might be unfair). Therefore, there are no data-sequencing dependencies between the components.

The request queue implements an intermediary that decouples the different modes of control from the HTTP handler (asynchronously) and from the Web server (synchronously). The queue itself offers a simple interface (`get` and `put`) that doesn't require a specific sequencing of control. Consequently, a control-sequencing dependency does not exist between the queue, HTTP handler, and Web server.

5.2.1.5 Quality-of-Service Dependencies

For this pattern, the quality of service does not apply. An HTTP message has to be processed, and there is no notion of a better or worse service. For example, if one component executes with more or less latency, none of the components have to be changed.

5.2.1.6 Resource-Behavior Dependencies

Both the HTTP handler and the Web servers share the request queue as a resource. They depend on the request queue's ability to serialize concurrent requests. The queue's inability to serialize would greatly impact the HTTP handler and the Web server. Therefore, the request queue has a resource-behavior dependency.

Table 3 shows the all the dependencies between the three modules. A plus sign indicates that the propagation might occur; a minus sign indicates that the propagation will not occur, because there is no dependency.

	Type of Change										
	Data Syntax	Data Semantics	Service Syntax	Service Semantics	Sequencing (Data)	Sequencing (Control)	Identity of an Interface	Existence	Location at Runtime	Quality of Service	Resource Behavior
Web server changes propagate to the request queue	+	-	+	+	-	-	+	+	+	-	+
HTTP handler changes propagate to the request queue	+	-	+	+	-	-	+	+	+	-	+
Web server changes propagate to the HTTP handler	+	+	-	-	-	-	-	+	-	-	-

Table 3: Dependencies Between the HTTP Handler, Web Server, and Request Queue

5.2.2 Analyzing the Wrapper Façade Pattern for Modifiability

An important JAWS requirement is that the system be portable to several OSs, especially UNIX, POSIX, and Windows. Many of the operating services, such as I/O, are already encapsulated in library functions used by the language runtime environment. Other OS services, such as process and thread management, sockets, dynamic linking, and time operations, are not. Although the OSs mentioned earlier provided those kinds of services, they each do so in different ways. Different syntax is required to access the services, which differ semantically.

The JAWS is implemented using object-orientation, while the interfaces to access OS services are procedural. This requires an adaptation between the different implementation styles. As a result, the JAWS uses the Wrapper Façade pattern to provide an object-oriented interface for OS services that is abstract enough to hide the implementation details of the supported OSs.

The pattern, as used in the JAWS, uses an intermediary to break service and data-syntax dependencies to the services provided by the OS. It also uses the tactic for separating the interface from the implementation to alleviate the semantic dependencies on the OS services. This is done by raising the abstraction level of the interface high enough to support the OSs and an OS-specific implementation. This implementation translates from the abstract interface to the details required by the OS. Figure 12 shows two possible configurations. Note, that the Web server application and the OS services interface stay the same, while the wrapper and the OS are different.

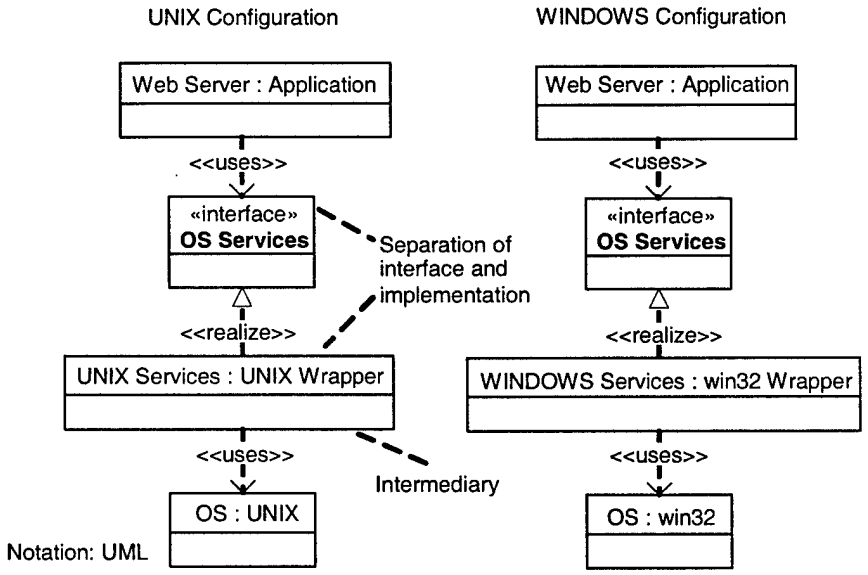


Figure 12: Two Possible Configurations Using the Wrapper Façade Pattern

5.2.2.1 Analysis of the Interface Abstraction Level

Analyzing the effect of separating the interface from the implementation would require a detailed interface specification of the OS services interface as well as the API specifications of the supported OSs. This detailed analysis is out of the scope of this report.

Nevertheless, even without the detailed specification, something can be said about that analysis. Clearly, the interface available to the Web server makes assumptions about the semantics of the services and data provided by the OSs. For example, the interface provides the concept of threads and services such as creating, deleting, suspending, and resuming threads. The interface was defined to support the UNIX and Windows NT OSs and abstracts away the differences between them. Therefore, changes to those OSs, such as upgrading to a new version, will not be visible through the interface. However, totally different OSs, especially those that do not understand the concept of threads, can't be supported without adaptations to the interface.

5.2.2.2 Analysis of the Dependencies

In terms of dependencies, the two configurations are equal. We used the configuration of UNIX to analyze the dependencies below.

Data- and Service-Syntax Dependencies

The UNIX services component adapts the data and service syntax presented through the OS services interface to the programming interface provided by the UNIX OS. Therefore, the Web server depends on the syntax provided by the UNIX services component, which, in turn, depends on the syntax provided by the OS. However, syntax dependencies no longer exist between the Web server and the OS services.

Data- and Service-Semantic Dependencies

The UNIX services component also translates the more abstract definition of data and service semantics provided by the OS services interface into the detailed interface definition of the specific OS—in this case, UNIX. This translation weakens, but does not remove, the semantic dependencies between the Web server and the OS. Of course, semantic dependencies also exist between the Web server and the UNIX services component, and between the UNIX services and the OS.

Data- and Control-Sequencing Dependencies

Without a detailed analysis of the OS services and the OS interface, it is not clear whether there is a dependency on data sequencing, meaning that several data items have to be exchanged in a certain sequence. However, the purpose of the UNIX services component is to hide differences between OSs. Therefore, dependencies in control sequencing should be hidden within the UNIX services component.

Identity, Existence, and Location Dependencies

The implementation of the Wrapper Façade pattern in the JAWS is done so that the Web server directly calls the wrapper UNIX services, which, in turn, directly call the API of the OS services. This interaction pattern requires that the Web server, the OS services component, and the UNIX API execute in the same process on the same processor.

Quality-of-Service and Resource-Behavior Dependencies

As before, the purpose of the UNIX services component is to hide OS-specific details from the applications. If different OSs behave differently and/or deliver their services in different qualities, the UNIX services have to cope with those differences. Therefore, there are no quality-of-service or resource-behavior dependencies between the Web server and the OS.

Table 4 shows the dependencies between the Web server, the UNIX services, and the OS. A plus sign indicates that the propagation might occur; a minus sign indicates that the propagation will not occur, because there is no dependency.

	Type of Change										
	Data Syntax	Data Semantics	Service Syntax	Service Semantics	Sequencing (Data)	Sequencing (Control)	Identity of an Interface	Existence	Location at Runtime	Quality of Service	Resource Behavior
Web server changes propagate to the UNIX services	+	+	+	+	+	+	+	+	+	+	+
Web server changes propagate to the OS	-	+[1]	-	+[1]	-	-	+	+	+	-	-
UNIX services changes propagate to the OS	+	+	+	+	+	+	+	+	+	+	+

[1] Semantic dependencies still exist, but they are weakened by the abstraction level of the OS services interface.

Table 4: Dependencies Between the Web Server, OS, and UNIX Services

In the row of Table 4 that shows the dependencies between the Web server and the OS, the UNIX services components acts as an intermediary to break the dependencies on data, service syntax, control sequencing, quality of service, and resource behavior. The use of the tactic for separating the interface from the implementation also weakens the semantic dependencies between the Web server and OS.

5.3 Applying Performance Tactics to Patterns

For the Half-Sync/Half-Async and Wrapper Façade patterns, we first identify the relevant performance tactics discussed in Section 4.4 and then draw on the analysis in Section 4.5 to describe some performance-related insights.

5.3.1 Applying Performance Tactics to the Half-Sync/Half-Async Pattern

The purpose of the Half-Sync/Half-Async pattern is to allow a Web server to simultaneously carry out several concurrent HTTP protocol sessions. Requests to carry out HTTP sessions are received by an I/O thread of the reactor and put into a request queue. Attention is then quickly returned to connection endpoints, so that the next request can be received. These requests are dequeued by a pool of threads for high-level protocol processing, where each thread handles a separate HTTP session. Each session involves getting a server-side file and sending it back to the client. The pattern is named as such because the high-level threads process the HTTP requests synchronously; however, each request is queued asynchronously by the low-level I/O thread.

As a first step in identifying tactics, we determine the resources involved, the sources of demand for them, and the scheduling policies for handling contention for them.

5.3.1.1 Resources

When considering resources, we look for processors, networks, disks, memory, and so forth. For the Half-Sync/Half-Async pattern, the main resource of concern is the processor (or processors) on which the Web server resides. Input to the Web server is coming from clients over a network, presumably through a network interface unit (NIU) and then into the server(s), so we will include the NIU as another resource. One of the main goals of the Web server is to get a hypertext markup language (HTML) file with associated data from a file server and send it back to the client. Therefore, the third relevant resource is a file server. These resources are shown in Figure 13.

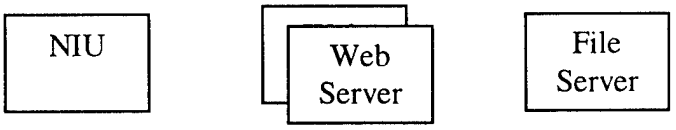


Figure 13: Resources Used in the Half-Sync/Half-Async Pattern

The architectural issues that are relevant to resource management tactics include

- opportunities to exploit physically concurrent processing
- properly allocating demand to multiple Web servers, if they are available

5.3.1.2 Demand

When considering demand, we look for event streams that arrive at the resources and that will result in resource utilization (e.g., CPU utilization or network bandwidth). Demand for half-sync/half-async resources is primarily in terms of client requests for the Web server.

Requests for Web servers originate randomly at one or more clients. Each request requires processing time on each of the above resources. As shown in Figure 14, processing here entails: the async layer acquiring input from the NIU and enqueuing connection or protocol commands; the queuing layer dequeuing commands; and the sync layer acquiring files from the file server and sending them back to the client.

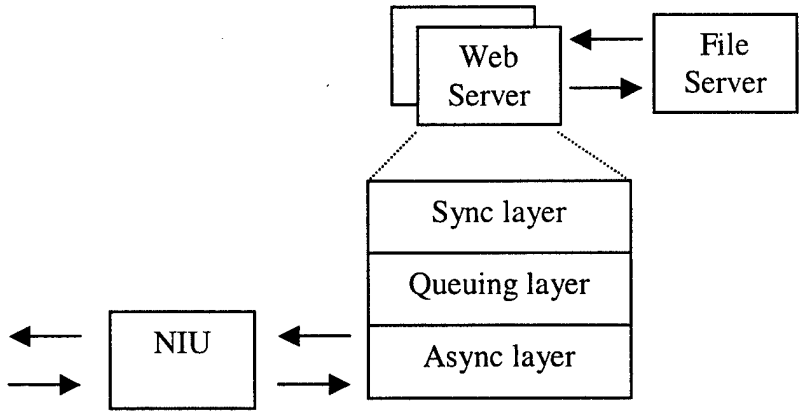


Figure 14: Processing Associated with Requests

Architectural issues relevant to resource management tactics include

- arrival-time and service-time distributions of requests
- variability of arrival and service times
- items that contribute to service time such as the processing time associated with each layer

5.3.1.3 Resource Scheduling

When considering resource scheduling, one looks for control threads, queues, scheduling policies, and sources of blocking. Messages (i.e., data-link-layer messages) arrive from the client over a network and are queued at the NIU. Within the OS kernel, an interrupt handler dequeues these messages and enqueues them in the TCP socket byte queue. When a message arrives, the application I/O thread (in the async layer) is activated. Then it reads the message and places it in the request queue (in the queuing layer). This execution must be performed as quickly as possible with as little interference from other processor threads as possible. The appropriate processing threads for the application request dequeue the HTTP messages for which they are responsible. The application threads interact with a file server to acquire files to send back to the client. Threads interact synchronously with the file server and, as a result, spend a significant portion of their time blocking. Two instances of the Half-Sync/Half-Async pattern are shown in Figure 15.

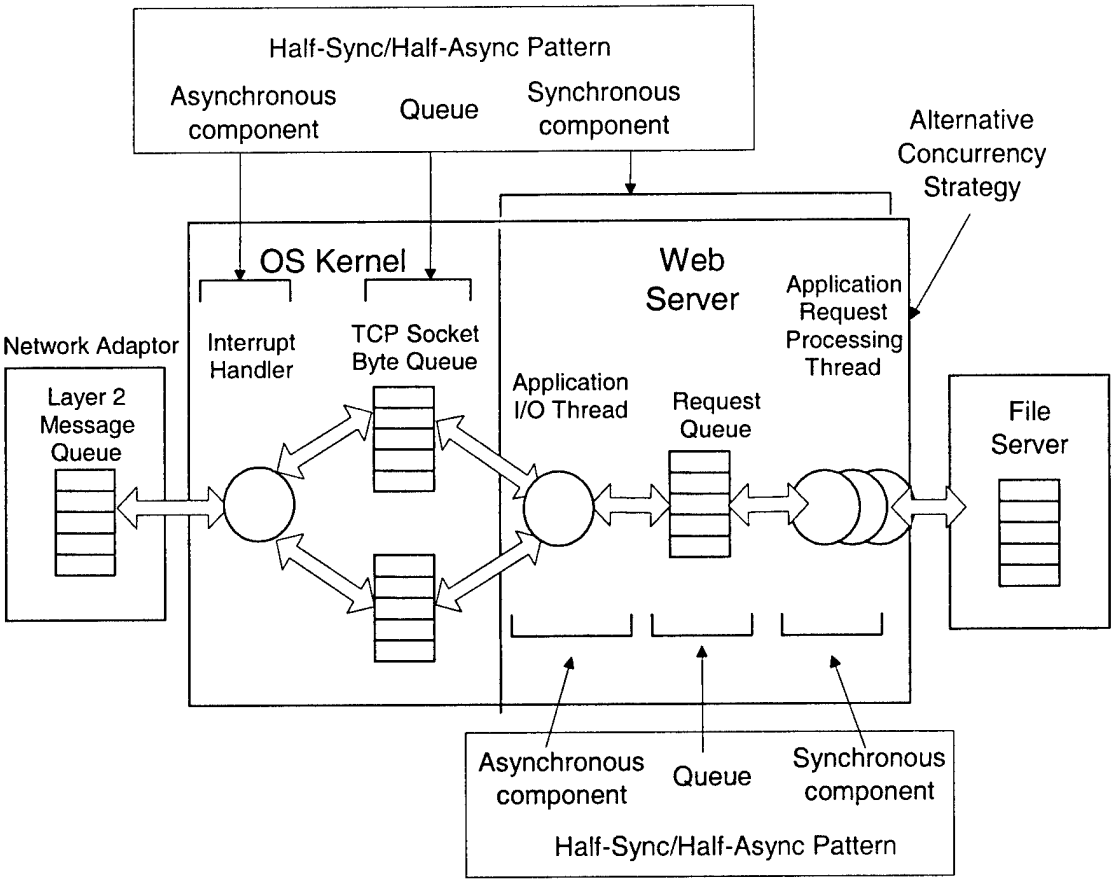


Figure 15: Two Instances of the Half-Sync/Half-Async Pattern

Figure 16 focuses on the Web server instance show in Figure 15.

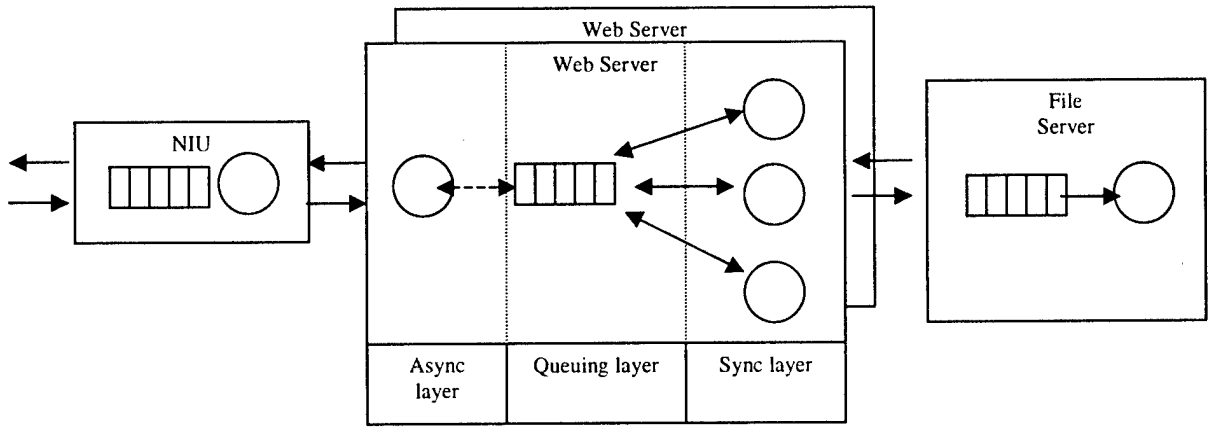


Figure 16: Web Server Instance

5.3.1.4 Performance Model

The analysis strategies for performance tactics are broken down into three areas: analysis of the effects of (1) queuing, (2) using multiple physical resources, and (3) scheduling. Each area is described below.

Queuing. The bold rectangles in Figure 17 highlight the queues and their associated servers. The leftmost rectangle contains the queue in the NIU and a process in the Web server. The async layer in the Web server is responsible primarily for dequeuing messages. The middle bold rectangle circumscribes the second queue and includes both the queuing and sync layers. Notice that several processes dequeue, but that only one is active at any one time. The others are blocking on I/O from the file server. The rightmost queue lives on the file server; its processor dequeues commands, finds files, and sends the files back synchronously to the process on the Web server.

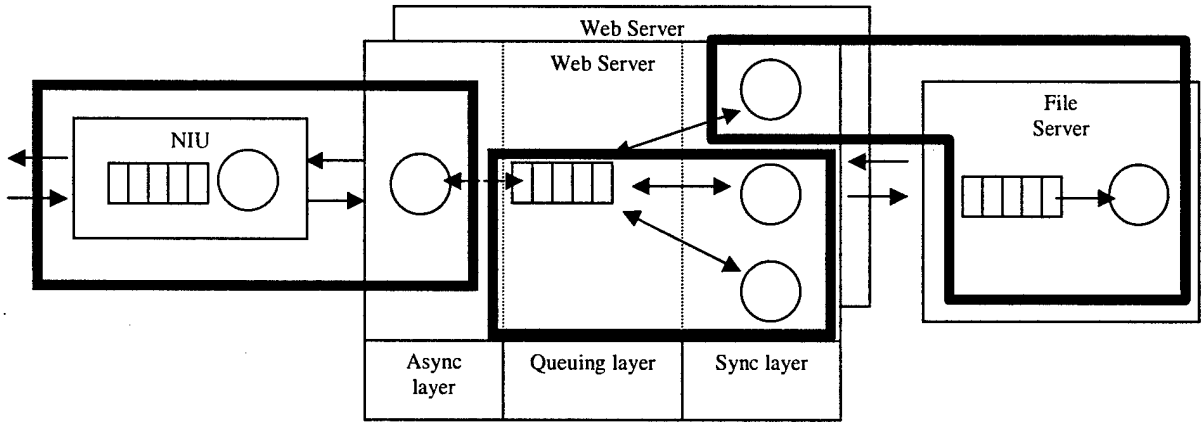


Figure 17: Queues with the Half-Sync/Half-Async Pattern

The tactics for controlling demand are directly applicable to each queue. For this high-level analysis, we assume that a single Web server is being used and that arrivals to each queue conform to the criteria for Poisson arrivals listed in Table 2 on page 33.

The first queue's execution time is assumed to be relatively short and constant. We also assume that the process on the NIU takes a negligible amount of time; that all of the work is done by the Web server process; and that the Web server process executes at a high priority relative to the other processes on the Web server. The graph in Figure 18 shows average latency as a function of utilization for both constant and exponential execution times when the average execution time is 1 ms.

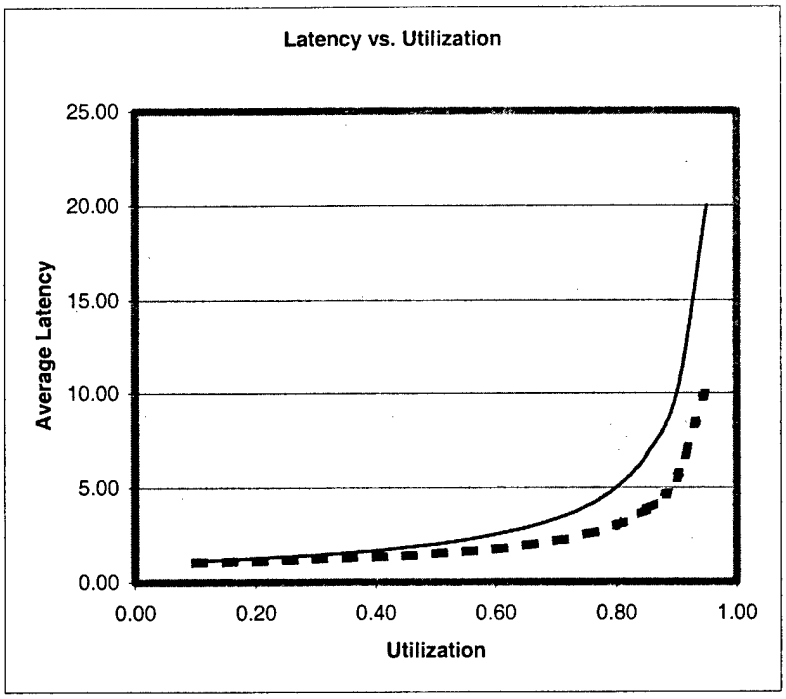


Figure 18: Graph Showing Average Latency as a Function of Utilization for Constant and Exponential Execution Times When the Average Execution Time is 1 ms

The graph above shows these two points: (1) latency increases significantly as a function of utilization, and (2) variability in execution times contributes to latency and is magnified as utilization increases. Because we don't expect the utilization of the first queue to be very high, we don't anticipate this to be a major contributor to latency.

In our subsequent analysis, we assume exponential arrival distribution. Accordingly, we use the following formula for our analysis:

$$E[T] = (U / 1-U) * E[S] + E[S]$$

Queue 1. We assume that the average time between arrivals is 1 second and the average execution time is 40 ms. So, the average latency is (.0417/(1-.0417))*40 + 40 = 42ms. In this case, we need to consider the following relevant tactics:

- bound variability—Variability seems naturally bounded, because we assume that the execution time is relatively short; the only duty of this server is to dequeue messages and then subsequently enqueue them.
- increase computational efficiency or reduce the computational overhead—Consider doubling the average execution time from 40 ms to 80 ms; the result will be slightly greater than doubling the average latency from 42 ms to 87 ms. As shown in Figure 18, the effect of increasing the execution time is exacerbated as utilization increases.

Queues 2 and 3. We assume that the average execution time of the second queue is 200 ms—the time it takes to dequeue the message, do some processing, and send a request for a file to the file server. This does not include the time it takes the file server to acquire the file (700 ms) and the time it takes to send the file back to the client (200 ms).

Next, we compare the two cases based on whether the physical concurrency is exploited. If we assume for a moment that the Web server uses a single thread to serve requests and that the thread blocks while the file server is fulfilling its request, the average execution time for the second queue is effectively 900 ms (200 + 700 + 200). In this case, the average latency for the Web server and data file together is 9000 ms.

When the physical concurrency is exploited and multiple threads are used, there is effectively one thread per customer in the file server's queue. This is because each thread dequeues a request and then makes a synchronous call to the data server. Assuming a sufficient number of threads, the execution time on the Web server is 200 ms resulting in an average latency on the Web server of 250 ms. The execution time on the Web server is 700 ms (500 + 200) with a resulting average latency of 2333 ms. The combined latency is 2583 ms, much less than when only a single thread is used. Note that we have not yet accounted for the fact that one of the processes associated with the third queue executes on the Web server.

Accounting for scheduling. If the process associated with queue 3 executes at a higher priority than the processes associated with queue 2, the execution time associated with serving queue 3 remains 700 ms (500 + 200). However, occasionally, the process serving queue 2 will be preempted for 200 ms by the process that is serving queue 3. To account for this, we inflate queue 2's average execution time from 200ms to 400ms. The resulting latency increases from 250ms to 667ms. Therefore, the combined latency increases from 2583ms to 3000ms.

Now let's account for contention for queue 2. Note that multiple processes access queue 2 and, while doing so, lock the queue. We assume that the time it takes to enqueue and dequeue messages is relatively short. However, the analysis of arbitration warns us that if the critical section is executed at the wrong priority, the blocking time can be relatively long even when the critical section is relatively short. Therefore, we also assume that the enqueue and dequeue services execute at a sufficiently high priority to avoid this problem.

Undoubtedly, this is not an exact queuing analysis. But, the point of tactics is to offer enough information to start building a model that will reveal many of the important factors that contribute to the event response. The model can then be refined as more detailed information becomes available.

5.3.2 Applying Performance Tactics to the Wrapper Façade Pattern

The purpose of the Wrapper Façade pattern is to avoid accessing low-level OS APIs directly. The example given in the JAWS case study is the Thread_Mutex Wrapper Façade, which provides a portable interface to OS mutual-exclusion mechanisms. Mutual-exclusion mechanisms, such as locks, semaphores, and monitors, are examples of the tactic for synchronizing protocols.

One of the key latency concerns is that enforcing mutual exclusion and managing it improperly may cause significant blocking. The abstract interface needs to capture this important attribute. While not all OSs support priority-inheritance protocols, the ability to control the priority at which critical sections are executed and to specify the synchronization protocol that is used highlights these protocols as important aspects of the interface. These aspects are just as important as *mutex_lock()*, *mutex_try_lock()*, and *mutex_unlock()*.

For example, it is important to know the priority at which the queue is locked—if it is locked at a priority that is too low, unnecessary blocking might result (as mentioned above). As another example, if the reactor executes at a high priority to dump requests into the queue, and if protocol threads execute at a lower priority to dequeue requests and carry out the protocol, there are other medium-priority threads doing other types of work. The medium-priority threads can prevent the requests from being dequeued and cause the reactor to block on the queue. This, in turn, could cause flow control to kick in and even slow down remote client requests.

6 Conclusions

Architectural tactics control the responses for particular attributes and are motivated using the analysis models for those attributes. Each attribute has its own collection of analysis models and tactics. Since tactics are important for controlling the achievement of quality, and since the achievement of quality drives architectural design and analysis, using the tactics discussed in this report is critical.

We have presented the tactics for modifiability and performance and shown how they contribute to the analysis of patterns or software architecture fragments. We view this report as the first in a series that identifies tactics and explores the issues associated with tactics. Future work in this area includes

- completing the tactics for and analyses of the modifiability and performance attributes. Only a partial list of each is discussed in this report.
- extending the analysis of tactics to other attributes. Security, availability, and usability are important attributes for which tactics and analysis models should also be identified.
- integrating tactics into architectural design methods such as the ADD method [Bachmann 02]. If tactics are the key to controlling attributes' responses and controlling those responses is the key to architecture design, it follows that tactics are the key to architectural design.

Regardless of the remaining work, we remain convinced that architectural tactics are the key to understanding how standard patterns and techniques achieve quality attributes. We also believe that tactics are the key to designing systems for which behavior can be predicted *prior* to the systems' construction.

Abbreviations, Acronyms, and Initialisms

ABAS	Attribute-Based Architectural Style
ADD	Attribute-Driven Design
API	application program interface
ATAM	Architecture Tradeoff Analysis Method
CPU	central processing unit
DM	deadline monotonic
EDF	earliest deadline first
FCFS	first come, first served
FIFO	first in, first out
HTML	hypertext markup language
HTTP	hypertext transfer protocol
I/O	input/output
JAWS	Java Adaptive Web Server
ms	milliseconds
NIU	network interface unit
OS	operating system
QAW	Quality Attribute Workshop
QoS	quality of service
QRAM	QoS Resource Allocation Method
RM	rate monotonic
RMA	Rate Monotonic Analysis
RMI	Remote Method Invocation

RPI	Remote Procedure Invocation
SEI	Software Engineering Institute
TCP/IP	Transmission Control Protocol/Internet Protocol
UML	Unified Modeling Language

References

[Bachmann 00] Bachmann, F.; Bass, L.; & Klein, M. *Quality Attribute Design Primitives* (CMU/SEI-2000-TN-017, ADA392284). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2000. <http://www.sei.cmu.edu/publications/documents/00.reports/00tn017.html>.

[Bachmann 02] Bachmann, F.; Bass, L.; & Klein, M. "Quality Attribute Design Primitives and the Attribute-Driven Design," 169-186. *Proceedings for the 4th International Workshop on Software Product Family Engineering*. Bilboa, Spain, October 3-5, 2001. Berlin, Germany: Springer-Verlag, 2002.

[Bass 01] Bass, L.; Klein, M.; & Moreno, G. *Applicability of General Scenarios to the Architecture Tradeoff Analysis Method* (CMU/SEI-2001-TR-014, ADA396098). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 2001. <http://www.sei.cmu.edu/publications/documents/01.reports/01tr014.html>.

[Buschmann 96] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; & Stal, M. *Pattern-Oriented Software Architecture*. Chichester, NY: Wiley, 1996.

[Buschmann 00] Buschmann, F.; Schmidt, D.; & Stal, M. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. New York, NY: J. Wiley & Sons, 2000.

[Gamma 95] Gamma, E.; Helm, R.; Johnson, R.; & Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Kazman 99] Kazman, R. & Klein, M. *Attribute-Based Architectural Styles* (CMU/SEI-99-TR-022, ADA371802). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1999. <http://www.sei.cmu.edu/publications/documents/99.reports /99tr022/99tr022abstract.html>.

[Lee 98] Lee, C.; Lehoczky, J. P.; Rajkumar, R.; & Siewiorek, D. P. "Practical Solutions for QoS-Based Resource Allocation Problems," 296-306. *Proceedings of the 19th IEEE Real-Time Systems Symposium.* Madrid, Spain, December 2-4, 1998. Los Alamitos, CA: IEEE Computer Society, 1998.

[Rajkumar 91] Rajkumar, R. *Synchronization in Real-Time Systems: A Priority Inheritance Approach.* Boston, MA: Kluwer Academic Publishers, 1991.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	August 2002	Final

4. TITLE AND SUBTITLE
Illuminating the Fundamental Contributors to Software Architecture Quality

5. FUNDING NUMBERS
F19628-00-C-0003

6. AUTHOR(S)
Felix Bachmann, Len Bass, Mark Klein

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

8. PERFORMING ORGANIZATION REPORT NUMBER
CMU/SEI-2002-TR-025

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
HQ ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2116

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
ESC-TR-2002-025

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified/Unlimited, DTIC, NTIS

12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

An architectural tactic is a design decision that helps achieve a specific quality-attribute response. Such a tactic must be motivated by a quality-attribute analysis model. This report presents the basic concepts of analysis models for two quality attributes—modifiability and performance, identifies a collection of tactics that can be used to control responses within those models, and discusses how to analyze the models in terms of these tactics.

This report also describes how to interpret architectural designs in terms of analysis models and how to apply those models to specific architectures. In addition, it presents the analysis of several different architectural patterns taken from current literature.

14. SUBJECT TERMS	15. NUMBER OF PAGES
software architecture, architectural tactics, architectural design, architecture analysis	76

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL